AD-A040 990 TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CALIF F/G 9/2
MINIMAL BASIC SEMANOL (76) SPECIFICATION LISTING.(U)
MAY 77 F C BELZ, R M HART, D M HEIMBIGNER F30602-76-C-0245
UNCLASSIFIED RADC-TR-77-170-VOL-2 NL

1 OF 2
ADA040990

RADC-TR-77-170, Volume II (of two)
Final Technical Report
May 1977

MINIMAL BASIC SEMANOL (76) SPECIFICATION LISTING

TRW Defense and Space Systems Group

Approved for public release; distribution unlimited.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

This report has been reviewed and is approved for publication.

APPROVED: John M. Ives

JOHN M. IVES, Captain, USAF
Project Engineer

APPROVED: Alan R. Barnum

ALAN R. BARNUM
Assistant Chief
Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-77-170, Volume II (of two)	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) MINIMAL BASIC SEMANOL (76) SPECIFICATION LISTING		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report. 27 Apr 76 — 27 Jan 77,
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) Frank C. Belz, Ruth M. Hart, Dennis M. Heimbigner		8. CONTRACT OR GRANT NUMBER(s) F30602-76-C-0245
9. PERFORMING ORGANIZATION NAME AND ADDRESS TRW Defense and Space Systems Group, One Space Park, Redondo Beach CA 90278		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 63728F 55500846
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441		12. REPORT DATE May 1977
		13. NUMBER OF PAGES 158
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)
Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer:
Captain John M. Ives (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
SEMANOL, SEMANOL (73), SEMANOL (76), BASIC, Minimal BASIC, BASIC Nucleus, language definition, standardization, semantics, syntax, language control, metalanguage, interpreter

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
This report contains a listing of the SEMANOL (76) metalanguage specification of the Minimal BASIC programming language. The specification is complete and has been extensively computer tested. The SEMANOL (76) metalanguage used here is a formal one that has an interpretive approach; its use has allowed implementation dependent semantics to be included in this specification.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

4Ø9637.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

```
#DECLARE-GLOBAL:

     active-for-block-list,
     basic-program,
     current-print-line,
     current-statement,
     data-list-pointer,
     first-time-through,
     initial-input-state,
     input-file,
     input-from-terminal,
     input-line,
     latest-return-point,
     return-point-list
     #.
```

```
#DECLARE-SYNTACTIC-COMPONENT:

     all-fors-have-matching-nexts-in,
     argument-expression-of,
     array-declaration-for,
     bounds-part-of,
     control-variable-in,
     def-statement-expression-of,
     def-statement-name-of,
     def-statement-parameter-of,
     def-statement-with-name,
     destination-line-number-list-in,
     destination-line-number-of,
     ends-in-separator,
     first-dimension-bound-of,
     first-dimension-of,
     first-dimension-upper-bound-value-for,
     first-executable-statement-starting-with,
     has-an-argument,
     has-one-dimension,
     has-two-dimensions,
     increment-part-of-for,
     index-expression-of,
     initial-value-part-of-for,
     input-data-list-in,
     is-abs-function-ref,
     is-atn-function-ref,
     is-cos-function-ref,
     is-def-statement-parameter,
     is-def-statement-with-parameter,
     is-executable-statement,
     is-exp-function-ref,
     is-explicitly-declared-array,
     is-int-function-ref,
     is-log-function-ref,
     is-non-executable,
     is-not-a-control-statement,
     is-not-stop-or-end,
     is-numeric-datum,
     is-numeric-defined-function-ref,
     is-numeric-expression,
     is-numeric-relational-expression,
     is-numeric-variable,
     is-parenthetical,
     is-print-separator,
     is-quoted-string,
     is-rnd-function-ref,
     is-sgn-function-ref,
```

```
is-simple-control-statement,
is-sin-function-ref,
is-sqr-function-ref,
is-string-constant,
is-string-expression,
is-string-relational-expression,
is-string-variable,
is-tan-function-ref,
last-seg-of,
left-hand-side-of,
limit-part-of-for,
line-containing,
line-number-part-of,
line-number-value-of,
list-of-variables-to-be-input-in,
matching-next,
nameable-part-of,
next-executable-statement-following,
next-statement-successor-of,
number-of-bounds-in,
number-of-dimensions-in,
number-of-subscripts-in,
numeric-array-name-of,
numeric-defined-function-name-of,
numeric-defined-function-ref-of,
numeric-expression-of,
numeric-supplied-function-ref-of,
operand-1-of,
operand-2-of,
option-base-for,
option-base-of,
parent-node,
print-list-sequence-of,
relation-of,
relational-expression-of,
right-hand-side-of,
root-node,
s-own-line-number,
second-dimension-bound-of,
second-dimension-of,
second-dimension-upper-bound-value-for,
sequence-of-ancestors-of,
sequence-of-array-declarations-and-references-in,
sequence-of-array-declarations-in,
sequence-of-array-references-in,
sequence-of-def-statements-in,
sequence-of-defined-function-references-in,
sequence-of-executable-statements-in,
```

```
sequence-of-for-statements-in,
sequence-of-for-statements-preceding,
sequence-of-line-ids-in,
sequence-of-lines-in,
sequence-of-next-statements-following,
sequence-of-next-statements-in,
sequence-of-option-statements-in,
sequence-of-statements-in,
simple-statement-successor-of,
standard-array-element-name-of,
standard-name-of,
standard-parameter-name-derived-from,
statement-containing,
statement-part-of,
statement-whose-line-number-is-equivalent-to,
string-constant-of,
string-expression-of,
string-variable-of,
subscript-part-of,
totality-of-data-in
#.
```

```
#CONTEXT-FREE-SYNTAX:

#DF program

     => <%<line>> <end-line> #.

#DF line

     => <line-id> <#GAP> <statement> <#GAP> <end-of-line> #.

#DF line-id

     => <line-number> #.

#DF end-of-line

     => <'[LF]'> <#GAP> #.

#DF end-line

     => <line-id> <#GAP> <end-statement> <#GAP>
        <end-of-line> #.

#DF end-statement

     => <'END'> #.

#DF statement

     => <data-statement>
     => <def-statement>
     => <dimension-statement>
     => <for-statement>
     => <gosub-statement>
     => <goto-statement>
     => <if-then-statement>
     => <input-statement>
     => <numeric-let-statement>
     => <string-let-statement>
     => <next-statement>
     => <on-goto-statement>
```

=> <option-statement>
=> <print-statement>
=> <randomize-statement>
=> <read-statement>
=> <remark-statement>
=> <restore-statement>
=> <return-statement>
=> <stop-statement> #.

#DF numeric-let-statement

=> <'LET'> <#GAP> <numeric-variable> <#GAP> <equals>
<#GAP> <numeric-expression> #.

#DF string-let-statement

=> <'LET'> <#GAP> <string-variable> <#GAP> <equals>
<#GAP> <string-expression> #.

#DF goto-statement

=> <'GO'> <%<#SPACE>> <'TO'> <#GAP> <line-number> #.

#DF line-number

=> <digit>
=> <digit> <digit>
=> <digit> <digit> <digit>
=> <digit> <digit> <digit> <digit> #.

#DF if-then-statement

=> <'IF'> <#GAP> <relational-expression> <#GAP>
<'THEN'> <#GAP> <line-number> #.

#DF relational-expression

=> <numeric-expression> <#GAP> <relation> <#GAP>
<numeric-expression>
=> <string-expression> <#GAP> <equality-relation>
<#GAP> <string-expression> #.

```
#DF relation

     => <equality-relation>
     => <less-than>
     => <greater-than>
     => <not-less>
     => <not-greater> #.


#DF equality-relation

     => <equals>
     => <not-equals> #.


#DF not-less

     => <greater-than> <equals> #.


#DF not-greater

     => <less-than> <equals> #.


#DF not-equals

     => <less-than> <greater-than> #.


#DF gosub-statement

     => <'GO'> <%<#SPACE>> <'SUB'> <#GAP> <line-number> #.


#DF return-statement

     => <'RETURN'> #.


#DF on-goto-statement

     => <'ON'> <#GAP> <numeric-expression> <#GAP> <'GO'>
        <%<#SPACE>> <'TO'> <#GAP> <line-number> <% <<#GAP>
        <comma> <#GAP> <line-number>>> #.
```

```
#DF stop-statement

     => <'STOP'> #.

#DF for-statement

     => <'FOR'> <#GAP> <control-variable> <#GAP> <equals>
        <#GAP> <initial-value> <#GAP> <'TO'> <#GAP> <limit>
     => <'FOR'> <#GAP> <control-variable> <#GAP> <equals>
        <#GAP> <initial-value> <#GAP> <'TO'> <#GAP> <limit>
        <#GAP> <'STEP'> <#GAP> <increment> #.

#DF control-variable

     => <simple-numeric-variable> #.

#DF initial-value

     => <numeric-expression> #.

#DF limit

     => <numeric-expression> #.

#DF increment

     => <numeric-expression> #.

#DF next-statement

     => <'NEXT'> <#GAP> <control-variable> #.

#DF print-statement

     => <'PRINT'> <print-list> #.

#DF print-list

     => <%<<#NILSET #U <<#GAP> <print-item>>> <#GAP>
        <print-separator>>> <#NILSET #U <<#GAP>
```

```
        <print-item>>> #.


#DF print-item

     => <expression>
     => <tab-call> #.


#DF tab-call

     => <'TAB'> <#GAP> <open> <#GAP> <numeric-expression>
        <#GAP> <close> #.


#DF print-separator

     => <comma>
     => <semicolon> #.


#DF end-of-print-line

     => #NILSET #.


#DF input-statement

     => <'INPUT'> <#GAP> <variable-list> #.


#DF variable-list

     => <variable> <%<<#GAP> <comma> <#GAP> <variable>>> #.


#DF data-statement

     => <'DATA'> <#GAP> <data-list> #.


#DF data-list

     => <datum> <%<<#GAP> <comma> <#GAP> <datum>>> #.


#DF datum
```

```
     => <quoted-string>
     => <unquoted-string> #.


#DF read-statement

     => <'READ'> <#GAP> <variable-list> #.


#DF restore-statement

     => <'RESTORE'> #.


#DF dimension-statement

     => <'DIM'> <#GAP> <array-declaration> <%<<#GAP> <comma>
        <#GAP> <array-declaration>>> #.


#DF array-declaration

     => <numeric-array-name> <#GAP> <open> <#GAP> <bounds>
        <#GAP> <close> #.


#DF bounds

     => <integer>
     => <integer> <#GAP> <comma> <#GAP> <integer> #.


#DF option-statement

     => <'OPTION BASE'> <#GAP> <'0' ,'1'> #.


#DF def-statement

     => <'DEF'> <#GAP> <numeric-defined-function> <#GAP>
        <equals> <#GAP> <numeric-expression>
     => <'DEF'> <#GAP> <numeric-defined-function> <#GAP>
        <parameter-list> <#GAP> <equals> <#GAP>
        <numeric-expression> #.


#DF numeric-defined-function
```

```
     => <'FN'> <letter> #.


#DF parameter-list

     => <open> <#GAP> <parameter> <#GAP> <close> #.


#DF parameter

     => <simple-numeric-variable> #.


#DF randomize-statement

     => <'RANDOMIZE'> #.


#DF remark-statement

     => <'REM'>
     => <'REM'> <#SPACE> <remark-string> #.


#DF expression

     => <string-expression>
     => <numeric-expression> #.


#DF numeric-expression

     => <term>
     => <positive-expression>
     => <negation>
     => <sum>
     => <difference> #.


#DF positive-expression

     => <plus> <#GAP> <term> #.


#DF negation

     => <minus> <#GAP> <term> #.
```

#DF sum

=> <numeric-expression> <#GAP> <plus> <#GAP> <term> #.

#DF difference

=> <numeric-expression> <#GAP> <minus> <#GAP> <term> #.

#DF term

=> <factor>
=> <product>
=> <quotient> #.

#DF product

=> <term> <#GAP> <asterisk> <#GAP> <factor> #.

#DF quotient

=> <term> <#GAP> <slant> <#GAP> <factor> #.

#DF factor

=> <primary>
=> <involution> #.

#DF involution

=> <factor> <#GAP> <circumflex> <#GAP> <primary> #.

#DF primary

=> <numeric-variable>
=> <numeric-rep>
=> <numeric-function-ref>
=> <open> <#GAP> <numeric-expression> <#GAP> <close> #.

#DF numeric-function-ref

```
     => <numeric-defined-function-ref>
     => <numeric-supplied-function-ref> #.


#DF numeric-defined-function-ref

     => <numeric-defined-function>
     => <numeric-defined-function> <#GAP> <argument-list> #.


#DF numeric-supplied-function-ref

     => <'ABS'> <#GAP> <argument-list>
     => <'ATN'> <#GAP> <argument-list>
     => <'COS'> <#GAP> <argument-list>
     => <'EXP'> <#GAP> <argument-list>
     => <'INT'> <#GAP> <argument-list>
     => <'LOG'> <#GAP> <argument-list>
     => <'RND'>
     => <'SGN'> <#GAP> <argument-list>
     => <'SIN'> <#GAP> <argument-list>
     => <'SQR'> <#GAP> <argument-list>
     => <'TAN'> <#GAP> <argument-list> #.


#DF argument-list

     => <open> <#GAP> <argument> <#GAP> <close> #.


#DF argument

     => <numeric-expression> #.


#DF string-expression

     => <string-variable>
     => <string-constant> #.


#DF constant

     => <numeric-constant>
     => <string-constant> #.
```

```
#DF numeric-constant

     => <#NILSET #U sign> <numeric-rep> #.


#DF sign

     => <plus>
     => <minus> #.


#DF numeric-rep

     => <significand> <#NILSET #U exrad> #.


#DF significand

     => <integer> <#NILSET #U period>
     => <#NILSET #U integer> <fraction> #.


#DF integer

     => <%1<digit>> #.


#DF fraction

     => <period> <%1<digit>> #.


#DF exrad

     => <'E'> <#NILSET #U sign> <integer> #.


#DF string-constant

     => <quoted-string> #.


#DF variable

     => <numeric-variable>
     => <string-variable> #.
```

```
#DF numeric-variable

     => <simple-numeric-variable>
     => <numeric-array-element> #.


#DF simple-numeric-variable

     => <letter> <#NILSET #U digit> #.


#DF numeric-array-element

     => <numeric-array-name> <#GAP> <subscript> #.


#DF numeric-array-name

     => <letter> #.


#DF subscript

     => <open> <#GAP> <numeric-expression> <#GAP> <close>
     => <open> <#GAP> <numeric-expression> <#GAP> <comma>
        <#GAP> <numeric-expression> <#GAP> <close> #.


#DF string-variable

     => <letter> <dollar> #.


#DF letter

     => <'A','B','C','D','E','F','G','H','I','J','K','L',
        'M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z'>
        #.


#DF digit

     => <'0','1','2','3','4','5','6','7','8','9'> #.


#DF string-character

     => <quote>
```

```
     => <quoted-string-character> #.


#DF quoted-string-character

     => <ampersand>
     => <apostrophe>
     => <comma>
     => <exclamation-point>
     => <unquoted-string-character> #.


#DF unquoted-string-character

     => <space>
     => <plain-string-character> #.


#DF plain-string-character

     => <letter>
     => <digit>
     => <asterisk>
     => <circumflex>
     => <close>
     => <colon>
     => <dollar>
     => <equals>
     => <greater-than>
     => <less-than>
     => <minus>
     => <number-sign>
     => <open>
     => <percent>
     => <period>
     => <plus>
     => <question-mark>
     => <semicolon>
     => <slant>
     => <underline> #.


#DF remark-string

     => <%<string-character>> #.


#DF quoted-string
```

```
     => <quote> <%<quoted-string-character>> <quote> #.


#DF unquoted-string

     => <plain-string-character>
     => <plain-string-character>
        <%<unquoted-string-character>>
        <plain-string-character> #.


#DF keyword

     => <'BASE','DATA','DEF','DIM','END',
        'FOR','GO','GOSUB','GOTO','IF',
        'INPUT','LET','NEXT','ON','OPTION',
        'PRINT','RANDOMIZE','READ','REM',
        'RESTORE','RETURN','STEP','STOP','SUB','THEN','TO'>
        #.


#DF space

     => <#SPACE> #.


#DF exclamation-point

     => <'!'> #.


#DF quote

     => <'"'> #.


#DF number-sign

     => <'#'> #.


#DF dollar

     => <'$'> #.


#DF percent
```

```
     => <'%'> #.

#DF ampersand

     => <'&'> #.

#DF apostrophe

     => <'[']'> #.

#DF open

     => <'('> #.

#DF close

     => <')'> #.

#DF asterisk

     => <'*'> #.

#DF plus

     => <'+'> #.

#DF comma

     => <','> #.

#DF minus

     => <'-'> #.

#DF period

     => <'.'> #.
```

```
#DF slant

     => <'/'> #.

#DF colon

     => <':'> #.

#DF semicolon

     => <';'> #.

#DF less-than

     => <'<'> #.

#DF equals

     => <'='> #.

#DF greater-than

     => <'>'> #.

#DF question-mark

     => <'?'> #.

#DF circumflex

     => <'^'> #.

#DF underline

     => <'_'> #.

#DF input-prompt
```

```
     => <'? '> #.

#DF input-reply

     => <data-list> <#GAP> <end-of-input-reply> #.

#DF end-of-input-reply

     => <'[LF]'> #.
```

```
#CONTROL-COMMANDS:

  #ASSIGN-VALUE! basic-program = #CONTEXT-FREE-PARSE-TREE
  (#GIVEN-PROGRAM, "wrt" <program> )

  #IF ($basic-program$) is-context-free-syntactically-valid
  #THEN

  #IF ($basic-program$) is-contextually-syntactically-valid
  #THEN

  #BEGIN

    #COMPUTE! initialize-globals

    #ASSIGN-VALUE! current-statement = #FIRST-ELEMENT-IN
    sequence-of-executable-statements-in(basic-program)

    #WHILE ($current-statement$) is-not-stop-or-end #DO

    #BEGIN

      #COMPUTE! effect-of(current-statement)

      #ASSIGN-VALUE! current-statement =
      statement-successor-of(current-statement)

    #END

    #IF current-print-line #NEQW #NIL
    #THEN #COMPUTE! print(end-of-print-line-char)

  #END

  #COMPUTE! #STOP #.
```

```
#SEMANTIC-DEFINITIONS:

#DF is-context-free-syntactically-valid(prog)

     "{prog #EQ basic-program}"

     => #TRUE #IF prog #IS-NOT #UNDEFINED ;

     =>
        false-due-to-error('context-free-syntax-error-in-program-text')
        #OTHERWISE #.

#PROC-DF false-due-to-error(msg)

     "{msg #IS #STRING}"

     #BEGIN

       #COMPUTE! fatal-syntactic-error('error: ' #CW msg)

       #RETURN-WITH-VALUE! #FALSE

     #END #.

#DF fatal-syntactic-error(msg)

     "{msg #IS #STRING}"

     => #OUTPUT(msg #CW end-of-print-line-char) #.

#DF is-contextually-syntactically-valid(prog)

     "{prog #EQ basic-program}"

     => #TRUE #IFF all-line-nrs-are-non-zero-in(prog) #AND
        lines-are-in-ascending-line-nr-order-in(prog) #AND
        lines-are-uniquely-numbered-in(prog) #AND
        all-line-numbers-exist-in(prog) #AND
        all-fors-have-matching-nexts-in(prog) #AND
        all-nexts-have-matching-fors-in(prog) #AND
        fors-and-nexts-are-properly-matched-in(prog) #AND
        arrays-are-uniquely-dimensioned-in(prog) #AND
        arrays-are-defined-first-in(prog) #AND
        consistent-number-of-subscripts-in(prog) #AND
```

```
        no-dimension-option-conflict(prog) #AND
        option-statement-is-first-in(prog) #AND
        functions-are-uniquely-defined-in(prog) #AND
        all-functions-are-defined-in(prog) #AND
        no-recursive-functions-in(prog) #AND
        functions-are-defined-first-in(prog) #AND
        consistent-number-of-arguments-in(prog) #.

#DF all-line-nrs-are-non-zero-in(prog)

     "{prog #EQ basic-program}"

     => #TRUE #IF #FOR-ALL line-nr #IN
        sequence-of-line-ids-in(prog) #IT-IS-TRUE-THAT
        (line-number-value-of(line-nr) #N= 0) ;

     => false-due-to-error('zero-valued-line-number')
        #OTHERWISE #.

#DF lines-are-in-ascending-line-nr-order-in(prog)

     "{prog #EQ basic-program}"

     => #TRUE #IF #FOR-ALL line-nr #IN
        all-but-last-element-in
        (sequence-of-line-ids-in(prog)) #IT-IS-TRUE-THAT
        (line-number-value-of(line-nr) <
        line-number-value-of(line-nr-next-following(line-nr)))
        ;

     => false-due-to-error('lines-out-of-order') #OTHERWISE
        #.

#DF all-but-last-element-in(list)

     "{list #IS #SEQUENCE}"

     => #INITIAL-SUBSEQ-OF-LENGTH(#LENGTH(list) - 1) #OF
        list #.

#DF line-nr-next-following(line-nr)

     "{line-nr #IS <line-id>}"
```

```
    => #FIRST ln #IN
       sequence-of-line-ids-in(root-node(line-nr))
       #SUCH-THAT (line-nr #PRECEDES ln #IN
       sequence-of-line-ids-in(root-node(line-nr))) #.


#DF lines-are-uniquely-numbered-in(prog)

    "{prog #EQ basic-program}"

    => #TRUE #IF #FOR-ALL ln1 #IN
       sequence-of-line-ids-in(prog) #IT-IS-TRUE-THAT
       (#FOR-ALL ln2 #IN sequence-of-line-ids-in(prog)
       #IT-IS-TRUE-THAT (line-number-value-of(ln1) =
       line-number-value-of(ln2) #IMPLIES ln1 #EQ ln2)) ;

    => false-due-to-error('duplicate-line-numbers')
       #OTHERWISE #.


#DF all-line-numbers-exist-in(prog)

    "{prog #EQ basic-program}"

    => #TRUE #IF #NOT #THERE-EXISTS stmt #IN
       sequence-of-executable-statements-in(prog)
       #SUCH-THAT (nonexistent-line-is-referenced-by(stmt))
       ;

    => false-due-to-error('nonexistent-line-number')
       #OTHERWISE #.


#DF nonexistent-line-is-referenced-by(stmt)

    "{($stmt$) is-basic-statement}"

    => nonexistent-line-is-referenced-by-on-goto(stmt) #IF
       stmt #IS <on-goto-statement> ;

    =>
       nonexistent-line-is-referenced-by-other-control(stmt)
       #IF stmt #IS <goto-statement> #U <gosub-statement>
       #U <if-then-statement> ;

    => #FALSE #OTHERWISE #.
```

```
#DF nonexistent-line-is-referenced-by-on-goto(stmt)

     "{stmt #IS <on-goto-statement>}"

     => #TRUE #IFF #THERE-EXISTS ln1 #IN
        destination-line-number-list-in(stmt) #SUCH-THAT
        (#FOR-ALL ln2 #IN sequence-of-line-ids-in
        (root-node(stmt)) #IT-IS-TRUE-THAT
        (line-number-value-of(ln1) #N=
        line-number-value-of(ln2))) #.


#DF nonexistent-line-is-referenced-by-other-control(stmt)

     "{stmt #IS <goto-statement> #U <gosub-statement> #U
       <if-then-statement>}"

     => #TRUE #IFF #FOR-ALL line-nr #IN
        sequence-of-line-ids-in(root-node(stmt))
        #IT-IS-TRUE-THAT (line-number-value-of(line-nr) #N=
        line-number-value-of(destination-line-number-of(stmt)))
        #.


#DF all-fors-have-matching-nexts-in(prog)

     "{prog #EQ basic-program}"

     => #TRUE #IF #FOR-ALL for-stmt #IN
        sequence-of-for-statements-in(prog) #IT-IS-TRUE-THAT
        (#THERE-EXISTS next-stmt #IN
        sequence-of-next-statements-in(prog) #SUCH-THAT
        (($for-stmt, "and" next-stmt$) match #AND #FOR-ALL
        other-for-stmt #IN
        sequence-of-for-statements-in(prog) #IT-IS-TRUE-THAT
        (($other-for-stmt, "and" next-stmt$) match #IMPLIES
        for-stmt #DOES-NOT-PRECEDE other-for-stmt #IN
        sequence-of-executable-statements-in(prog)))) ;

     =>
        false-due-to-error('for-statement-has-no-matching-next')
        #OTHERWISE #.


#DF match(for-stmt, "and" next-stmt)

     "{for-stmt #IS <for-statement> #AND next-stmt #IS
     <next-statement>}"
```

```
     => #TRUE #IFF
        #STRING-OF-TERMINALS-OF(control-variable-in
        (for-stmt)) #EQW
        #STRING-OF-TERMINALS-OF(control-variable-in
        (next-stmt)) #AND for-stmt #PRECEDES next-stmt #IN
        sequence-of-executable-statements-in(root-node(for-stmt))
        #.


#DF all-nexts-have-matching-fors-in(prog)

     "{prog #EQ basic-program}"

     => #TRUE #IF #FOR-ALL next-stmt #IN
        sequence-of-next-statements-in(prog)
        #IT-IS-TRUE-THAT (#THERE-EXISTS for-stmt #IN
        sequence-of-for-statements-in(prog) #SUCH-THAT
        (($for-stmt, "and" next-stmt$) match #AND #FOR-ALL
        other-next-stmt #IN
        sequence-of-next-statements-in(prog)
        #IT-IS-TRUE-THAT (($for-stmt, "and"
        other-next-stmt$) match #IMPLIES other-next-stmt
        #DOES-NOT-PRECEDE next-stmt #IN
        sequence-of-executable-statements-in(prog)))) ;

     =>
        false-due-to-error('next-statement-has-no-matching-for')
        #OTHERWISE #.


#DF fors-and-nexts-are-properly-matched-in(prog)

     "{prog #EQ basic-program}"

     => #FALSE #IF #NOT
        all-fors-have-matching-nexts-in(prog) ;

     => #TRUE #IF #FOR-ALL stmt1 #IN
        sequence-of-for-statements-in (prog)
        #IT-IS-TRUE-THAT (#FOR-ALL stmt2 #IN
        sequence-of-for-statements-in(prog) #IT-IS-TRUE-THAT
        (($stmt2, "in" stmt1$) is-nested #IMPLIES
        ($matching-next(stmt2), "in" stmt1$) is-nested)) ;

     => false-due-to-error('improperly-nested-for-blocks')
        #OTHERWISE #.
```

```
#DF is-nested(stmt2, "in" stmt1)

     "{stmt1 #IS <for-statement> #AND stmt2 #IS
     <for-statement>}"

     => #TRUE #IFF stmt1 #PRECEDES stmt2 #IN
        sequence-of-executable-statements-in(root-node(stmt2))
        #AND stmt2 #PRECEDES matching-next(stmt1) #IN
        sequence-of-executable-statements-in(root-node(stmt2))
        #.

#DF arrays-are-uniquely-dimensioned-in(prog)

     "{prog #EQ basic-program}"

     => #TRUE #IF #FOR-ALL array-decl-1 #IN
        sequence-of-array-declarations-in(prog)
        #IT-IS-TRUE-THAT (#FOR-ALL array-decl-2 #IN
        sequence-of-array-declarations-in(prog)
        #IT-IS-TRUE-THAT
        (#STRING-OF-TERMINALS-OF(numeric-array-name-of(array-decl-1))
        #EQW
        #STRING-OF-TERMINALS-OF(numeric-array-name-of(array-decl-2))
        #IMPLIES array-decl-1 #EQ array-decl-2)) ;

     => false-due-to-error('multiply-defined-array')
        #OTHERWISE #.

#DF arrays-are-defined-first-in(prog)

     "{prog #EQ basic-program}"

     => #TRUE #IF #FOR-ALL array-decl #IN
        sequence-of-array-declarations-in(prog)
        #IT-IS-TRUE-THAT (#FOR-ALL array-ref #IN
        sequence-of-array-references-in(prog)
        #IT-IS-TRUE-THAT
        (#STRING-OF-TERMINALS-OF(numeric-array-name-of(array-decl))
        #EQW
        #STRING-OF-TERMINALS-OF(numeric-array-name-of(array-ref))
        #IMPLIES line-containing(array-decl) #PRECEDES
        line-containing(array-ref) #IN
        sequence-of-lines-in(prog))) ;

     =>
```

```
     false-due-to-error('array-referenced-before-declaration')
     #OTHERWISE #.


#DF consistent-number-of-subscripts-in(prog)

    "{prog #EQ basic-program}"

    => #TRUE #IF #FOR-ALL array-1 #IN
       sequence-of-array-declarations-and-references-in(prog)
       #IT-IS-TRUE-THAT (#FOR-ALL array-2 #IN
       sequence-of-array-declarations-and-references-in(prog)
       #IT-IS-TRUE-THAT
       (#STRING-OF-TERMINALS-OF(numeric-array-name-of(array-1))
       #EQW
       #STRING-OF-TERMINALS-OF(numeric-array-name-of(array-2))
       #IMPLIES number-of-dimensions-in(array-1) #EQ
       number-of-dimensions-in(array-2))) ;

    => false-due-to-error('array-subscript-inconsistency')
       #OTHERWISE #.


#DF number-of-dimensions-in(node)

    "{node #IS <numeric-array-element> #U
    <array-declaration>}"

    => number-of-subscripts-in(node) #IF node #IS
       <numeric-array-element> ;

    => number-of-bounds-in(node) #IF node #IS
       <array-declaration> #.


#DF number-of-subscripts-in(node)

    "{node #IS <numeric-array-element>}"

    => 1 #IF subscript-part-of(node) #IS #CASE 1 #OF
       <subscript> ;

    => 2 #IF subscript-part-of(node) #IS #CASE 2 #OF
       <subscript> #.


#DF number-of-bounds-in(node)
```

```
     "{node #IS <array-declaration>}"

     => 1 #IF bounds-part-of(node) #IS #CASE 1 #OF <bounds>
        ;

     => 2 #IF bounds-part-of(node) #IS #CASE 2 #OF <bounds>
        #.


#DF no-dimension-option-conflict(prog)

     "{prog #EQ basic-program}"

     => #TRUE #IF #FOR-ALL stmt #IN
        sequence-of-option-statements-in(prog)
        #IT-IS-TRUE-THAT (option-base-of(stmt) = 1 #IMPLIES
        #NOT #THERE-EXISTS array-decl #IN
        sequence-of-array-declarations-in(prog) #SUCH-THAT
        (($bounds-part-of(array-decl)$)
        has-a-zero-upper-bound)) ;

     => false-due-to-error('dimension-option-conflict')
        #OTHERWISE #.


#DF has-a-zero-upper-bound(b)

     "{b #IS <bounds>}"

     => #TRUE #IFF #STRING-OF-TERMINALS-OF
        (first-dimension-bound-of(b)) #EQ 0 #IF ($b$)
        has-one-dimension ;

     => #TRUE #IFF #STRING-OF-TERMINALS-OF
        (first-dimension-bound-of(b)) #EQ 0 #OR
        #STRING-OF-TERMINALS-OF
        (second-dimension-bound-of(b)) #EQ 0 #IF ($b$)
        has-two-dimensions #.


#DF option-statement-is-first-in(prog)

     "{prog #EQ basic-program}"

     => #TRUE #IF #FOR-ALL stmt #IN
        sequence-of-option-statements-in(prog)
        #IT-IS-TRUE-THAT (#FOR-ALL array-ref #IN
        sequence-of-array-declarations-and-references-in(prog)
```

```
        #IT-IS-TRUE-THAT (line-containing(stmt) #PRECEDES
        line-containing(array-ref) #IN
        sequence-of-lines-in(prog))) ;

     =>
        false-due-to-error('option-statement-follows-array-reference')
        #OTHERWISE #.


#DF functions-are-uniquely-defined-in(prog)

     "{prog #EQ basic-program}"

     => #TRUE #IF #FOR-ALL stmt1 #IN
        sequence-of-def-statements-in(prog) #IT-IS-TRUE-THAT
        (#FOR-ALL stmt2 #IN
        sequence-of-def-statements-in(prog) #IT-IS-TRUE-THAT
        (#STRING-OF-TERMINALS-OF(def-statement-name-of(stmt1))
        #EQW
        #STRING-OF-TERMINALS-OF(def-statement-name-of(stmt2))
        #IMPLIES stmt1 #EQ stmt2)) ;

     => false-due-to-error('multiply-defined-function')
        #OTHERWISE #.


#DF all-functions-are-defined-in(prog)

     "{prog #EQ basic-program}"

     => #TRUE #IF #FOR-ALL fn #IN
        sequence-of-defined-function-references-in(prog)
        #IT-IS-TRUE-THAT
        (numeric-defined-function-name-of(fn) #IS
        <numeric-defined-function> #IMPLIES #THERE-EXISTS
        stmt #IN sequence-of-def-statements-in(prog)
        #SUCH-THAT (#STRING-OF-TERMINALS-OF
        (def-statement-name-of(stmt)) #EQW
        #STRING-OF-TERMINALS-OF
        (numeric-defined-function-name-of(fn)))) ;

     => false-due-to-error('missing-function-definition')
        #OTHERWISE #.


#DF no-recursive-functions-in(prog)

     "{prog #EQ basic-program}"
```

```
    => #TRUE #IF #FOR-ALL stmt #IN
       sequence-of-def-statements-in(prog) #IT-IS-TRUE-THAT
       (#FOR-ALL fn #IN
       sequence-of-defined-function-references-in(prog)
       #IT-IS-TRUE-THAT (line-containing(fn) #EQ
       line-containing(stmt) #IMPLIES
       #STRING-OF-TERMINALS-OF
       (numeric-defined-function-name-of(fn)) #NEQW
       #STRING-OF-TERMINALS-OF
       (def-statement-name-of(stmt)))) ;

    => false-due-to-error('recursive-function-definition')
       #OTHERWISE #.


#DF functions-are-defined-first-in(prog)

    "{prog #EQ basic-program}"

    => #TRUE #IF #FOR-ALL stmt #IN
       sequence-of-def-statements-in(prog) #IT-IS-TRUE-THAT
       (#FOR-ALL fn #IN
       sequence-of-defined-function-references-in(prog)
       #IT-IS-TRUE-THAT (#STRING-OF-TERMINALS-OF
       (numeric-defined-function-name-of(fn)) #EQW
       #STRING-OF-TERMINALS-OF
       (def-statement-name-of(stmt)) #IMPLIES
       line-containing(fn) #DOES-NOT-PRECEDE
       line-containing(stmt) #IN
       sequence-of-lines-in(prog))) ;

    =>
       false-due-to-error('function-referenced-before-definition')
       #OTHERWISE #.


#DF consistent-number-of-arguments-in(prog)

    "{prog #EQ basic-program}"

    => #TRUE #IF #FOR-ALL stmt #IN
       sequence-of-def-statements-in(prog) #IT-IS-TRUE-THAT
       (all-function-references-agree-with(stmt)) ;

    =>
       false-due-to-error('inconsistent-number-of-arguments')
       #OTHERWISE #.
```

```
#DF all-function-references-agree-with(stmt)

     "{stmt #IS <def-statement>}"

     => ($stmt$) references-have-no-arguments #IF stmt #IS
        #CASE 1 #OF <def-statement> ;

     => ($stmt$) references-have-one-argument #IF stmt #IS
        #CASE 2 #OF <def-statement> #.


#DF references-have-no-arguments(stmt)

     "{stmt #IS #CASE 1 #OF <def-statement>}"

     => #TRUE #IFF #FOR-ALL fn #IN
        sequence-of-defined-function-references-in(root-node(stmt))
        #IT-IS-TRUE-THAT (#STRING-OF-TERMINALS-OF
        (numeric-defined-function-name-of(fn)) #EQW
        #STRING-OF-TERMINALS-OF
        (def-statement-name-of(stmt)) #IMPLIES fn #IS #CASE
        1 #OF <numeric-defined-function-ref> ) #.


#DF references-have-one-argument(stmt)

     "{stmt #IS #CASE 2 #OF <def-statement>}"

     => #TRUE #IFF #FOR-ALL fn #IN
        sequence-of-defined-function-references-in(root-node(stmt))
        #IT-IS-TRUE-THAT (#STRING-OF-TERMINALS-OF
        (numeric-defined-function-name-of(fn)) #EQW
        #STRING-OF-TERMINALS-OF
        (def-statement-name-of(stmt)) #IMPLIES fn #IS #CASE
        2 #OF <numeric-defined-function-ref> ) #.


#DF sequence-of-lines-in(prog)

     "{prog #EQ basic-program}"

     => #SEQUENCE-OF <line> #IN prog #.


#DF sequence-of-line-ids-in(prog)
```

```
     "{prog #EQ basic-program}"

     => #SEQUENCE-OF <line-id> #IN prog #.

#DF sequence-of-for-statements-in(prog)

     "{prog #EQ basic-program}"

     => #SEQUENCE-OF <for-statement> #IN prog #.

#DF sequence-of-next-statements-in(prog)

     "{prog #EQ basic-program}"

     => #SEQUENCE-OF <next-statement> #IN prog #.

#DF sequence-of-array-declarations-in(prog)

     "{prog #EQ basic-program}"

     => #SEQUENCE-OF <array-declaration> #IN prog #.

#DF sequence-of-array-references-in(prog)

     "{prog #EQ basic-program}"

     => #SEQUENCE-OF <numeric-array-element> #IN prog #.

#DF sequence-of-array-declarations-and-references-in(prog)

     "{prog #EQ basic-program}"

     => sequence-of-array-declarations-in(prog) #CS
        sequence-of-array-references-in(prog) #.

#DF sequence-of-option-statements-in(prog)

     "{prog #EQ basic-program}"

     => #SEQUENCE-OF <option-statement> #IN prog #.
```

```
#DF sequence-of-def-statements-in(prog)

     "{prog #EQ basic-program}"

     => #SEQUENCE-OF <def-statement> #IN prog #.

#DF sequence-of-defined-function-references-in(prog)

     "{prog #EQ basic-program}"

     => #SEQUENCE-OF <numeric-defined-function-ref> #IN prog
        #.
```

```
#PROC-DF initialize-globals

  #BEGIN

    #ASSIGN-VALUE! data-list-pointer = 1

    #ASSIGN-VALUE! return-point-list = #NILSEQ

    #ASSIGN-VALUE! active-for-block-list = #NILSEQ

    #ASSIGN-VALUE! current-print-line = #NIL

    #ASSIGN-VALUE! first-time-through = #TRUE

    #ASSIGN-VALUE! initial-input-state = #TRUE

    #RETURN-WITH-VALUE! #NIL

  #END #.

#DF is-not-stop-or-end(stmt)

     "{stmt #EQ current-statement}"

     => #TRUE #IFF #NOT stmt #IS <stop-statement> #U
        <end-statement> #.

#DF effect-of(stmt)

     "{stmt #EQ current-statement}"

     => #NIL #IF ($stmt$)is-non-executable #OR
        ($stmt$)is-simple-control-statement ;

     => for-statement-effect(stmt) #IF stmt #IS
        <for-statement> ;

     => gosub-statement-effect(stmt) #IF stmt #IS
        <gosub-statement> ;

     => input-statement-effect(stmt) #IF stmt #IS
        <input-statement> ;

     => numeric-let-statement-effect(stmt) #IF stmt #IS
```

```
        <numeric-let-statement> ;

     => string-let-statement-effect(stmt) #IF stmt #IS
        <string-let-statement> ;

     => next-statement-effect(stmt) #IF stmt #IS
        <next-statement> ;

     => print-statement-effect(stmt) #IF stmt #IS
        <print-statement> ;

     => read-statement-effect(stmt) #IF stmt #IS
        <read-statement> ;

     => restore-statement-effect #IF stmt #IS
        <restore-statement> #.


#DF for-statement-effect(stmt)

     "{stmt #IS <for-statement> }"

     => reset-first-time-through #IF #NOT first-time-through
        ;

     => activate-for-block(stmt) #OTHERWISE #.


#PROC-DF reset-first-time-through

  #BEGIN

    #ASSIGN-VALUE! first-time-through = #TRUE

    #RETURN-WITH-VALUE! #NIL

  #END #.


#PROC-DF activate-for-block(stmt)

     "{stmt #IS <for-statement> }"

     #BEGIN

       "deactivate any for block with the same control
       variable"
```

```
        #IF #THERE-EXISTS x #IN active-for-block-list
        #SUCH-THAT (control-variable-is-active(x,"in" stmt))
        #THEN #ASSIGN-VALUE! active-for-block-list =
        new-active-for-block-list(stmt)

        "check for maximum number of active for blocks"

        #IF #LENGTH(active-for-block-list) >=
        max-number-of-for-blocks
        #THEN

        #COMPUTE!
        fatal-error('too-many-for-blocks-active-at-one-time')

        "activate the current for block"

        #ASSIGN-VALUE! active-for-block-list =
        \for-block-list-element(stmt)\ #CS
        active-for-block-list

        "initialize the control variable"

        #COMPUTE! #ASSIGN-LATEST-VALUE
        (standard-name-of(control-variable-in(stmt)),"receives"
        initial-value-in-for(stmt))

        #RETURN-WITH-VALUE! #NIL

    #END #.

#DF initial-value-in-for(stmt)

    "{stmt #IS <for-statement> }"

    => numeric-value(initial-value-part-of-for(stmt)) #.

#DF value-of-limit-in-for(stmt)

    "{stmt #IS <for-statement> }"

    => numeric-value(limit-part-of-for(stmt)) #.

#DF value-of-increment-in-for(stmt)

    "{stmt #IS <for-statement> }"
```

```
     => numeric-value(increment-part-of-for(stmt)) #IF stmt
           #IS #CASE 2 #OF <for-statement> ;

     => implementation-one #OTHERWISE #.

#DF new-active-for-block-list(stmt)

     "{stmt #IS <for-statement> }"

     => first-part-of(active-for-block-list,"up to"
           active-control-variable(stmt)) #CS
           second-part-of(active-for-block-list,"after"
           active-control-variable(stmt)) #.

#DF active-control-variable(stmt)

     "{stmt #IS <for-statement> #U <next-statement> }"

     => #FIRST x #IN active-for-block-list #SUCH-THAT
           (control-variable-is-active(x,"in" stmt)) #.

#DF control-variable-is-active(x,"in" stmt)

     "{x #IS for-block-list-element
       & stmt #IS <for-statement> #U <next-statement> }"

     => #TRUE #IFF standard-name-of(#FIRST-ELEMENT-IN x) #EQ
           standard-name-of(control-variable-in(stmt)) #.

#DF first-part-of(list,"up to for-block-list-element" x)

     "{list #EQ active-for-block-list
       & x #IS for-block-list-element}"

     =>
           #INITIAL-SUBSEQ-OF-LENGTH(position-of-control-variable(x)
           - 1) #OF list #.

#DF second-part-of(list,"after for-block-list-element" x)

     "{list #EQ active-for-block-list
       & x #IS for-block-list-element}"
```

```
     => #TERMINAL-SUBSEQ-OF-LENGTH(#LENGTH(list) -
        position-of-control-variable(x)) #OF list #.


#DF position-of-control-variable(x)

     "{x #IS for-block-list-element}"

     => #ORDPOSIT x #IN active-for-block-list #.


#DF for-block-list-element(stmt)

     "{stmt #IS <for-statement> }"

     => \control-variable-in(stmt),
        value-of-limit-in-for(stmt),
        value-of-increment-in-for(stmt)\ #.


#DF gosub-statement-effect(stmt)

     "{stmt #IS <gosub-statement> }"

     =>
        set-latest-return-point-to(simple-statement-successor-of(stmt))
        #.


#PROC-DF set-latest-return-point-to(stmt)

     "{ ($stmt$) is-basic-statement}"

     #BEGIN

       #IF #LENGTH(return-point-list) >=
       max-number-of-unreturned-gosubs
       #THEN #COMPUTE!
       fatal-error('too-many-unreturned-gosub-executions')

       "otherwise ..."

       #ASSIGN-VALUE! return-point-list = \stmt\ #CS
       return-point-list

       #RETURN-WITH-VALUE! #NIL
```

```
    #END #.

#PROC-DF input-statement-effect(stmt)

    "{stmt #IS <input-statement> }"

    #BEGIN

      #COMPUTE! #OUTPUT(input-prompt-character)

      #ASSIGN-VALUE! input-line =
      input-reply-tree(next-input-line)

      #COMPUTE! validate-input-data-for(stmt)

      #RETURN-WITH-VALUE! #NIL

    #END #.

#DF input-prompt-character

    "{#ON-RETURN: input-prompt-character #IS <input-prompt>
    }"

    => '? ' #.

#DF input-reply-tree(i-f-t)

    "{i-f-t #EQ input-from-terminal}"

    => #CONTEXT-FREE-PARSE-TREE(i-f-t,"wrt" <input-reply>)
       #.

#PROC-DF next-input-line

  #BEGIN

    #IF initial-input-state #THEN

    #COMPUTE! read-input-file

    #ASSIGN-VALUE! input-from-terminal = (#PREFIX-OF-FIRST
    end-of-input-reply-char #IN input-file) #CW
    end-of-input-reply-char
```

```
    #COMPUTE! #OUTPUT(input-from-terminal)

    #ASSIGN-VALUE! input-file = #SUFFIX-OF-FIRST
    end-of-input-reply-char #IN input-file

    #RETURN-WITH-VALUE! input-from-terminal

  #END #.

#PROC-DF read-input-file

  #BEGIN

    #ASSIGN-VALUE! input-file = #INPUT

    #ASSIGN-VALUE! initial-input-state = #FALSE

    #RETURN-WITH-VALUE! #NIL

  #END #.

#DF end-of-input-reply-char

     "{#ON-RETURN: end-of-input-reply-char #IS
     <end-of-input-reply>}"

     => '[LF]' #.

#DF validate-input-data-for(stmt)

     "{stmt #IS <input-statement> }"

     => input-new-data-for(stmt) #IF ($stmt$)
        is-invalid-input-reply ;

     => assign-input-values(stmt) #OTHERWISE #.

#PROC-DF assign-input-values(stmt)

     "{stmt #IS <input-statement> }"

     #BEGIN
```

```
    #FOR-ALL i : 1 <= i <=
    #LENGTH(list-of-variables-to-be-input-in(stmt)) #DO

    #BEGIN

      #COMPUTE! #ASSIGN-LATEST-VALUE(standard-name-of
      (list-element(i,"in"
      list-of-variables-to-be-input-in(stmt))),
      "receives" value-of-datum (list-element(i, "in"
      input-data-list-in(input-line)), "wrt"
      list-element(i,"in"
      list-of-variables-to-be-input-in(stmt))))

    #END

    #RETURN-WITH-VALUE! #NIL

  #END #.


#DF list-of-variables-to-be-input-in(stmt)

  "{stmt #IS <input-statement> #U <read-statement> }"

  => #SEQUENCE-OF <variable> #IN stmt #.


#DF list-element(number,"in" list)

  "{number #IS #INTEGER &
    list #IS #SEQUENCE}"

  => number #TH-ELEMENT-IN list #.


#DF value-of-datum(d, "wrt" var)

  "{d #IS <datum> & var #IS <variable> }"

  => remove-quotes-from (#STRING-OF-TERMINALS-OF(d)) #IF
     ($d$) is-quoted-string ;

  => numeric-representation-or-zero
     (#STRING-OF-TERMINALS-OF(d)) #IF ($var$)
     is-numeric-variable;

  => #STRING-OF-TERMINALS-OF(d) #OTHERWISE #.
```

```
#DF numeric-representation-or-zero(str)

     "{str #IS #STRING}"

     => numeric-constant-underflow-effect #IF ($str$)
        results-in-numeric-conversion-underflow ;

     => implementation-numeric-representation(str)
        #OTHERWISE #.


#DF input-data-list-in(ln)

     "{ln #EQ input-line}"

     => #SEQUENCE-OF <datum> #IN ln #.


#DF is-invalid-input-reply(stmt)

     "{stmt #IS <input-statement> }"

     => invalid-input-reply ('unrecognizable-input-reply')
        #IF input-line #IS #UNDEFINED ;

     => invalid-input-reply
        ('incorrect-number-of-data-items') #IF #NOT
        exactly-enough-data("wrt" stmt) ;

     => invalid-input-reply
        ('character-datum-for-numeric-variable') #IF #NOT
        input-data-types-match("wrt" stmt) ;

     => #NOT all-data-is-in-range("wrt" stmt) #OTHERWISE #.


#DF invalid-input-reply(msg)

     "{msg #IS #STRING}"

     => #TRUE #IF non-fatal-error(msg #CW
        ';please-reenter-data') #EQW #NIL #.


#DF exactly-enough-data("wrt" stmt)

     "{stmt #IS <input-statement> }"
```

```
      => #TRUE #IFF
         #LENGTH(list-of-variables-to-be-input-in(stmt)) =
         #LENGTH(input-data-list-in(input-line)) #.


#DF input-data-types-match("wrt" stmt)

      "{stmt #IS <input-statement> }"

      => #FALSE #IFF #THERE-EXISTS x : 1 <= x <=
         #LENGTH(list-of-variables-to-be-input-in(stmt))
         #SUCH-THAT ( ($ list-element(x,"in"
         list-of-variables-to-be-input-in(stmt)) $)
         is-numeric-variable & #NOT ($ list-element(x, "in"
         input-data-list-in(input-line)) $) is-numeric-datum
         ) #.


#DF is-numeric-datum(d)

      "{d #IS <datum> }"

      => #TRUE #IFF #CONTEXT-FREE-PARSE-TREE(d, "wrt"
         <numeric-constant>) #IS-NOT #UNDEFINED #.


#DF all-data-is-in-range("wrt" stmt)

      "{stmt #IS <input-statement> }"

      => #FALSE #IFF #THERE-EXISTS x : 1 <= x <=
         #LENGTH(input-data-list-in(input-line)) #SUCH-THAT
         (($ list-element(x, "in"
         input-data-list-in(input-line)), "wrt"
         list-element(x, "in"
         list-of-variables-to-be-input-in(stmt)) $)
         is-not-in-range) #.


#DF is-not-in-range(d, "wrt" v)

      "{d #IS <datum> & v #IS <variable>}"

      => string-value-is-not-in-range(d) #IF #NOT ($d$)
         is-numeric-datum ;

      => numeric-value-is-not-in-range(d) #IF ($v$)
```

```
      is-numeric-variable ;

   => string-value-is-not-in-range(d) #OTHERWISE #.


#DF numeric-value-is-not-in-range(d)

   "{($d$) is-numeric-datum}"

   => invalid-input-reply('numeric-datum-is-not-in-range')
      #IF ($d$) results-in-numeric-conversion-overflow ;

   => #FALSE #OTHERWISE #.


#DF string-value-is-not-in-range(d)

   "{d #IS <datum> & #NOT ($d$) is-numeric-datum}"

   => invalid-input-reply('string-datum-is-not-in-range')
      #IF ($ #STRING-OF-TERMINALS-OF(d) $)
      results-in-string-overflow ;

   => #FALSE #OTHERWISE #.


#PROC-DF input-new-data-for(stmt)

   "{stmt #IS input-statement}"

   #BEGIN

     #COMPUTE! #OUTPUT(input-prompt-character)

     #ASSIGN-VALUE! input-line =
     input-reply-tree(next-input-line)

     #COMPUTE! validate-input-data-for(stmt)

     #RETURN-WITH-VALUE! #NIL

   #END #.


#PROC-DF numeric-let-statement-effect(stmt)

   "{stmt #IS <numeric-let-statement> }"
```

```
    #BEGIN

      #COMPUTE! #ASSIGN-LATEST-VALUE(standard-name-of
      (left-hand-side-of(stmt)), "receives"
      numeric-value(right-hand-side-of(stmt)))

      #RETURN-WITH-VALUE! #NIL

    #END #.


#DF string-let-statement-effect(stmt)

    "{stmt #IS <string-let-statement> }"

    => short-string-let-statement-effect(stmt) #IF #NOT ($
       string-value(right-hand-side-of(stmt)) $)
       results-in-string-overflow ;

    => fatal-error('maximum-string-length-exceeded')
       #OTHERWISE #.


#PROC-DF short-string-let-statement-effect(stmt)

    "{stmt #IS <string-let-statement> &
      #LENGTH(string-value(right-hand-side-of(stmt))) <=
      max-assignable-string-length}"

    #BEGIN

      #COMPUTE! #ASSIGN-LATEST-VALUE(standard-name-of
      (left-hand-side-of(stmt)), "receives"
      string-value(right-hand-side-of(stmt)))

      #RETURN-WITH-VALUE! #NIL

    #END #.


#DF next-statement-effect(stmt)

    "{stmt #IS <next-statement> }"

    => increment-control-variable(stmt) #IF ($stmt$)
       matches-active-for ;

    => no-matching-active-for #OTHERWISE #.
```

```
#PROC-DF increment-control-variable(stmt)

     "{stmt #IS <next-statement> }"

     #BEGIN

       #ASSIGN-VALUE! first-time-through = #FALSE

       #COMPUTE! #ASSIGN-LATEST-VALUE(standard-name-of
       (control-variable-in(stmt)),"receives"
       perform(#LATEST-VALUE(standard-name-of(control-variable-in(stmt)))
       '+', increment-of-matching-for(stmt)))

       #RETURN-WITH-VALUE! #NIL

     #END #.

#DF increment-of-matching-for(stmt)

     "{stmt #IS <next-statement> }"

     => 3 #TH-ELEMENT-IN (active-control-variable(stmt)) #.

#DF matches-active-for(stmt)

     "{stmt #IS <next-statement> }"

     => #TRUE #IFF #THERE-EXISTS x #IN active-for-block-list
        #SUCH-THAT (control-variable-is-active(x,"in" stmt))
        #.

#DF no-matching-active-for

     => fatal-error('next-statement-matches-no-active-for')
        #.

#PROC-DF print-statement-effect (stmt)

     "{stmt #IS <print-statement> }"

     #BEGIN
```

```
        #FOR-ALL i : 1 <= i <= #LENGTH
        (print-list-sequence-of (stmt)) #DO

        #COMPUTE! convert-and-print (i #TH-ELEMENT-IN
        print-list-sequence-of (stmt))

        #IF #NOT ($stmt$) ends-in-separator
        #THEN #COMPUTE! print (end-of-print-line-char)

        #RETURN-WITH-VALUE! #NIL

      #END #.


#DF ends-in-separator(stmt)

      "{stmt #IS <print-statement> }"

      => #FALSE #IF print-list-sequence-of(stmt) #EQ #NILSEQ
         ;

      => #TRUE #IF ($ #LAST-ELEMENT-IN
         print-list-sequence-of(stmt) $) is-print-separator ;

      => #FALSE #OTHERWISE #.


#DF print-list-sequence-of (stmt)

      "{stmt #IS <print-statement> }"

      => #SEQUENCE-OF <expression> #U <tab-call> #U
         <print-separator> #IN stmt #.


#DF is-print-separator (nx)

      "{nx #IS <expression> #U <tab-call> #U
      <print-separator> }"

      => #TRUE #IFF nx #IS <print-separator> #.


#DF end-of-print-line-char

      "{#ON-RETURN: end-of-print-line-char #IS
      <end-of-print-line> }"
```

```
     => '[LF]' #.

#DF print (str)

     "{str #IS #STRING}"

     => append-and-output (str) #IF end-of-print-line-char
        #IS #SUBWORD str;

     => append-to-current-print-line (str) #OTHERWISE #.

#PROC-DF append-and-output (str)

     "{str #IS #STRING & end-of-print-line-char #IS #SUBWORD
     str}"

     #BEGIN

       #COMPUTE! append-to-current-print-line
       ((#PREFIX-OF-FIRST end-of-print-line-char #IN str)
       #CW end-of-print-line-char)

       #COMPUTE! output-current-print-line

       #COMPUTE! print (#SUFFIX-OF-FIRST
       end-of-print-line-char #IN str)

       #RETURN-WITH-VALUE! #NIL

     #END #.

#PROC-DF append-to-current-print-line (str)

     "{str #IS #STRING}"

     #BEGIN

       #ASSIGN-VALUE! current-print-line =
       current-print-line #CW str

       #RETURN-WITH-VALUE! #NIL

     #END #.
```

```
#PROC-DF output-current-print-line

  #BEGIN

    #COMPUTE! #OUTPUT (current-print-line)

    #ASSIGN-VALUE! current-print-line = #NIL

    #RETURN-WITH-VALUE! #NIL

  #END #.


#DF convert-and-print (x)

     "{x #IS <expression> #U <tab-call> #U <print-separator>
     }"

     => print-tab (tab-value (x)) #IF x #IS <tab-call> ;

     => #NIL #IF x #IS #CASE 2 #OF <print-separator> ;

     => print-comma #IF x #IS #CASE 1 #OF <print-separator>
        ;

     => print-the-item
        (implementation-string-output-representation
        (string-value (string-expression-of (x)))) #IF ($x$)
        is-string-expression;

     => print-the-item (numeric-output-representation
        (numeric-value (numeric-expression-of (x)))) #IF
        ($x$) is-numeric-expression #.


#DF tab-value (tc)

     "{tc #IS <tab-call> }"

     => ($ integer-value(numeric-expression-of(tc)) $)
        adjusted-for-tabbing #.


#DF adjusted-for-tabbing(n)

     "{n #IS #INTEGER}"

     => tab-value-less-than-one #IF n < 1 ;
```

```
     => residue(n - 1, "modulo" implementation-margin) + 1
        #OTHERWISE #.

#DF tab-value-less-than-one

     => 1 #IF non-fatal-error('tab-value-is-less-than-1;1
        assumed') #EQW #NIL #.

#DF residue (n,"modulo" m)

     "{n >= 0 & m>0}"

     => n - (n / m) * m #.

#DF integer-value (nx)

     "{nx #IS <numeric-expression> }"

     => ($($numeric-value (nx)$)rounded-to-an-integer$)
        converted-to-semanol-integer #.

#DF print-tab (n)

     "{1 <= n & n <= implementation-margin}"

     => print (($n - columnar-position$) spaces) #IF n >=
        columnar-position ;

     => print (end-of-print-line-char #CW ($n - 1$)
        spaces)#OTHERWISE #.

#DF columnar-position

     => #LENGTH (current-print-line) +1 #.

#DF spaces (n)

     "{0 <=n & n <= implementation-margin}"

     => #LEFT n #CHARACTERS-OF line-of-spaces #.
```

```
#DF line-of-spaces

     "{ON-RETURN: #LENGTH (line-of-spaces) =
     implementation-margin & #FOR-ALL x : 1 <= x <=
     implementation-margin #IT-IS-TRUE-THAT (x
     #TH-CHARACTER-IN line-of-spaces #IS #SPACE }"

     => blanks(implementation-margin) #.


#DF blanks(n)

     "{n #IS #INTEGER}"

     => #NIL #IF n <= 0 ;

     => #SPACE #CW blanks(n - 1) #OTHERWISE #.


#DF print-comma

     => print-tab (next-zone-tab-position) #IF #NOT
        already-in-last-print-zone;

     => print (end-of-print-line-char) #OTHERWISE #.


#DF next-zone-tab-position

     => #FIRST pos #IN list-of-zone-tab-positions #SUCH-THAT
        (pos > columnar-position) #IF #THERE-EXISTS pos #IN
        list-of-zone-tab-positions #SUCH-THAT (pos >
        columnar-position);

     => 1 #OTHERWISE #.


#DF list-of-zone-tab-positions

     => sequence-of-integers-of-length (nr-zones-in-margin
        ,"starting-with" 1 ,"in-steps-of"
        implementation-print-zone-width) #.


#DF sequence-of-integers-of-length (l ,"starting-at" i
,"in-steps-of" j)
```

```
    "{l >= 0 & i #IS #INTEGER & j #IS #INTEGER}"

    => #NILSEQ #IF l = 0 ;

    => \i\ #CS sequence-of-integers-of-length (l - 1
       ,"starting-at" i + j ,"in-steps-of" j) #OTHERWISE #.


#DF nr-zones-in-margin

    => implementation-margin /
       implementation-print-zone-width #IF residue
       (implementation-margin ,"modulo"
       implementation-print-zone-width) = 0 ;

    => implementation-margin /
       implementation-print-zone-width + 1 #OTHERWISE #.


#DF already-in-last-print-zone

    => #TRUE #IFF columnar-position >= #LAST-ELEMENT-IN
       list-of-zone-tab-positions #.


#DF print-the-item (str)

    "{str #IS #STRING}"

    => print (($str$) altered-if-too-long) #.


#DF altered-if-too-long (str)

    "{str #IS #STRING}"

    => str #IF #LENGTH (str) <= (implementation-margin + 1)
       - columnar-position;

    => end-of-print-line-char #CW margin-checked (str)
       #OTHERWISE #.


#DF margin-checked (str)

    "{str #IS #STRING}"

    => str #IF #LENGTH (str) <= implementation-margin;
```

```
    => #LEFT implementation-margin #CHARACTERS-OF str #CW
       end-of-print-line-char #CW margin-checked
       (#SUFFIX-OF-FIRST (#LEFT implementation-margin
       #CHARACTERS-OF str) #IN str) #OTHERWISE #.


#PROC-DF read-statement-effect(stmt)

    "{stmt #IS <read-statement> }"

    #BEGIN

      "is there enough data in the remainder of the data
      sequence?"

      #IF #LENGTH(list-of-variables-to-be-input-in(stmt)) >
      #LENGTH(totality-of-data-in(basic-program)) -
      data-list-pointer + 1
      #THEN

      #COMPUTE!
      fatal-error('not-enough-data-left-in-data-list')

      "assign data to variables in the read statement, if
      type matches"

      #FOR-ALL x : 1 <= x <=
      #LENGTH(list-of-variables-to-be-input-in(stmt)) #DO

      #BEGIN

        "is a string datum being assigned to a numeric
        variable?"

        #IF list-element(x,"in"
        list-of-variables-to-be-input-in(stmt)) #IS #CASE 1
        #OF <variable> & #NOT ($
        list-element(data-list-pointer, "in"
        totality-of-data-in(basic-program)) $)
        is-numeric-datum #THEN

        #COMPUTE!
        fatal-error('string-datum-assigned-to-numeric-variable')

        "assign datum to variable"

        #COMPUTE! assign-next-datum ("to" list-element(x,
```

```
        "in" list-of-variables-to-be-input-in(stmt)))

        "increment data-list-pointer"

        #ASSIGN-VALUE! data-list-pointer =
        data-list-pointer + 1

      #END

      #RETURN-WITH-VALUE! #NIL

    #END #.


#DF assign-next-datum("to" v)

    "{v #IS <variable> }"

    => assign-string-value-or-error ("to" v) #IF #NOT ($v$)
       is-numeric-variable ;

    => #ASSIGN-LATEST-VALUE(standard-name-of(v), "receives"
       numeric-constant-overflow-error-effect
       (#STRING-OF-TERMINALS-OF
       (list-element(data-list-pointer, "in"
       totality-of-data-in(basic-program))))) #IF ($
       list-element(data-list-pointer, "in"
       totality-of-data-in (basic-program)) $)
       results-in-numeric-conversion-overflow ;

    => #ASSIGN-LATEST-VALUE(standard-name-of(v), "receives"
       value-of-datum (list-element(data-list-pointer, "in"
       totality-of-data-in(basic-program)), "wrt" v))
       #OTHERWISE #.


#DF assign-string-value-or-error("to" v)

    "{v #IS <variable> }"

    => fatal-error ('string-datum-is-not-in-range') #IF ($
       list-element(data-list-pointer, "in"
       totality-of-data-in(basic-program)) $)
       results-in-string-overflow ;

    => #ASSIGN-LATEST-VALUE(standard-name-of(v), "receives"
       value-of-datum (list-element(data-list-pointer, "in"
       totality-of-data-in(basic-program)), "wrt" v))
```

```
        #OTHERWISE #.

#DF totality-of-data-in(prog)

     "{prog #EQ basic-program}"

     => #SEQUENCE-OF <datum> #IN prog #.

#PROC-DF restore-statement-effect

  #BEGIN

    #ASSIGN-VALUE! data-list-pointer = 1

    #RETURN-WITH-VALUE! #NIL

  #END #.

#DF statement-successor-of(stmt)

     "{stmt #EQ current-statement}"

     => simple-statement-successor-of(stmt) #IF ($stmt$)
        is-not-a-control-statement ;

     => goto-statement-successor-of(stmt) #IF stmt #IS
        <goto-statement> #U <gosub-statement> ;

     => if-then-statement-successor-of(stmt) #IF stmt #IS
        <if-then-statement> ;

     => on-goto-statement-successor-of(stmt) #IF stmt #IS
        <on-goto-statement> ;

     => for-statement-successor-of(stmt) #IF stmt #IS
        <for-statement> ;

     => next-statement-successor-of(stmt) #IF stmt #IS
        <next-statement> ;

     => return-statement-successor #IF stmt #IS
        <return-statement> #.

#DF simple-statement-successor-of (stmt)
```

```
     "{($stmt$) is-not-a-control-statement #OR stmt #IS
     <for-statement> #OR stmt #IS <if-then-statement> #OR
     stmt #IS gosub-statement}"

     => next-executable-statement-following (stmt) #.


#DF next-executable-statement-following (stmt)

     "{stmt #IS statement}"

     => #FIRST stmnt #IN
        sequence-of-executable-statements-in (basic-program)
        #SUCH-THAT (stmt #PRECEDES stmnt #IN
        sequence-of-statements-in (basic-program)) #.


#DF sequence-of-executable-statements-in (px)

     "{px #EQ basic-program }"

     => #SUBSEQUENCE-OF-ELEMENTS stmt #IN
        sequence-of-statements-in (px) #SUCH-THAT ( ($stmt$)
        is-executable-statement ) #.


#DF sequence-of-statements-in (px)

     "{px #EQ basic-program }"

     => #SEQUENCE-OF <data-statement>
                     #U <def-statement>
                     #U <dimension-statement>
                     #U <for-statement>
                     #U <gosub-statement>
                     #U <goto-statement>
                     #U <if-then-statement>
                     #U <input-statement>
                     #U <numeric-let-statement>
                     #U <string-let-statement>
                     #U <next-statement>
                     #U <on-goto-statement>
                     #U <option-statement>
                     #U <print-statement>
                     #U <randomize-statement>
                     #U <read-statement>
                     #U <remark-statement>
```

```
                    #U <restore-statement>
                    #U <return-statement>
                    #U <stop-statement>
                    #U <end-statement> #IN px #.

#DF is-executable-statement (stmt)

     "{stmt #EQ current-statement}"

     => #NOT is-non-executable (stmt) #.

#DF if-then-statement-successor-of (stmt)

     "{stmt #IS <if-then-statement>}"

     => first-executable-statement-starting-with
        (statement-whose-line-number-is-equivalent-to
        (destination-line-number-of (stmt))) #IF
        relation-value (relational-expression-of (stmt)) ;

     => simple-statement-successor-of (stmt) #OTHERWISE #.

#DF goto-statement-successor-of (stmt)

     "{stmt #IS <goto-statement> #U <gosub-statement> }"

     => first-executable-statement-starting-with
        (statement-whose-line-number-is-equivalent-to
        (destination-line-number-of (stmt))) #.

#DF statement-whose-line-number-is-equivalent-to (sn)

     "{sn #IS <line-number> }"

     => #FIRST stmt #IN sequence-of-statements-in
        (basic-program) #SUCH-THAT (line-number-value-of
        (($stmt$)s-own-line-number) = line-number-value-of
        (sn)) #.

#DF first-executable-statement-starting-with (stmt)

     "{($stmt$)is-basic-statement}"
```

```
     => stmt #IF ($stmt$) is-executable-statement;

     => next-executable-statement-following (stmt)
        #OTHERWISE #.


#DF s-own-line-number (stmt)

     "{($stmt$)is-basic-statement}"

     => line-number-part-of (line-containing (stmt)) #.


#DF line-number-value-of (n)

     "{n #IS <line-number> #U <line-id> }"

     => ($n$) with-leading-zeroes-suppressed #.


#DF with-leading-zeroes-suppressed (n)

     "{n #IS #STRING & (#LENGTH (n) >= 1 &
     #FIRST-CHARACTER-IN (n) #NEQW '-' #OR #LENGTH (n) >=
     2)}"

     => #SUBSTRING-OF-CHARACTERS index-of-first-non-zero-in
        (n) #TO #LENGTH (n) #OF n #IF first-character-in (n)
        #NEQW '-' ;

     => '-' #CW ($magnitude(n)$)
        with-leading-zeroes-suppressed #OTHERWISE #.


#DF index-of-first-non-zero-in (n)

     "{n #IS #STRING & #LENGTH (n) >= 1}"

     => #FIRST i : 1 <= i <= #LENGTH(n) #SUCH-THAT ( i
        #TH-CHARACTER-IN n #NEQW '0' #OR i #EQ #LENGTH(n))
        #.


#DF magnitude(n)

     "{n #IS #INTEGER}"

     => n #IF first-character-in (n) #NEQW '-' ;
```

```
     => #SUFFIX-OF-FIRST '-' #IN n #OTHERWISE #.


#DF line-containing (node)

     "{node #IS #NODE}"

     => #FIRST x #IN sequence-of-ancestors-of(node)
        #SUCH-THAT (x #IS <line> ) #.


#DF on-goto-statement-successor-of (stmt)

     "{stmt #IS <on-goto-statement> }"

     => statement-selected-by (integer-value
        (index-expression-of (stmt)) ,"from"
        destination-line-number-list-in (stmt)) #.


#DF statement-selected-by (ix ,"from" lnlist)

     "{ix #IS #INTEGER & lnlist #IS #SEQUENCE & #FOR-ALL ln
     #IN lnlist #IT-IS-TRUE-THAT (ln #IS line-number)}"

     => fatal-error
        ('on-goto-expression-value-less-than-one') #IF ix <
        1 ;

     => fatal-error
        ('on-goto-expression-value>line-nr-list-length') #IF
        ix > #LENGTH (lnlist) ;

     => first-executable-statement-starting-with
        (statement-whose-line-number-is-equivalent-to (ix
        #TH-ELEMENT-IN lnlist)) #OTHERWISE #.


#DF destination-line-number-list-in (stmt)

     "{stmt #IS <on-goto-statement> }"

     => #SEQUENCE-OF <line-number> #IN stmt #.


#DF for-statement-successor-of(stmt)
```


01/28/77
SEMANOL Project

```
     "{stmt #IS <for-statement> }"

     => deactivate-for-block(stmt) #IF ($stmt$)
        satisfies-for-expression;

     => simple-statement-successor-of(stmt) #OTHERWISE #.


#DF satisfies-for-expression(stmt)

     "{stmt #IS <for-statement> }"

     => #TRUE #IFF implementation-greater-than-test (perform
        (perform (#LATEST-VALUE (standard-name-of
        (control-variable-in(stmt))), '-', 2 #TH-ELEMENT-IN
        active-control-variable(stmt)), '*',
        modified-sign-of (increment-of-matching-for(stmt))),
        implementation-zero) #.


#DF modified-sign-of(inc)

     "{inc #IS implementation-number}"

     => implementation-one #IF implementation-not-less-test
        (inc,implementation-zero) ;

     => implementation-negative-one #OTHERWISE #.


#PROC-DF deactivate-for-block(stmt)

     "{stmt #IS <for-statement> }"

     #BEGIN

       #ASSIGN-VALUE! active-for-block-list =
       new-active-for-block-list(stmt)

       #RETURN-WITH-VALUE! simple-statement-successor-of
       (matching-next(stmt))

     #END #.


#DF matching-next(stmt)

     "{stmt #IS <for-statement> }"
```

```
    => #FIRST stmnt #IN
       sequence-of-next-statements-following (stmt)
       #SUCH-THAT
       (standard-name-of(control-variable-in(stmnt)) #EQ
       standard-name-of(control-variable-in(stmt))) #.


#DF sequence-of-next-statements-following(stmt)

    "{stmt #IS <for-statement> }"

    => #SUBSEQUENCE-OF-ELEMENTS stmnt #IN
       sequence-of-next-statements-in (root-node(stmt))
       #SUCH-THAT (stmt #PRECEDES stmnt #IN
       root-node(stmt)) #.


#DF next-statement-successor-of(stmt)

    "{stmt #IS <next-statement> }"

    => #LAST stmnt #IN sequence-of-for-statements-preceding
       (stmt) #SUCH-THAT
       (standard-name-of(control-variable-in(stmnt)) #EQ
       standard-name-of(control-variable-in(stmt))) #.


#DF sequence-of-for-statements-preceding(stmt)

    "{stmt #IS <next-statement> }"

    => #SUBSEQUENCE-OF-ELEMENTS stmnt #IN
       sequence-of-for-statements-in (root-node(stmt))
       #SUCH-THAT (stmnt #PRECEDES stmt #IN
       root-node(stmt)) #.


#DF return-statement-successor

    "{current-statement #IS <return-statement> }"

    => retrieve-latest-return-point #.


#PROC-DF retrieve-latest-return-point

    "{current-statement #IS <return-statement> }"
```

```
    #BEGIN

      #IF return-point-list #EQ #NILSEQ #THEN

      #COMPUTE!
      fatal-error('attempt-to-execute-more-returns-than-gosubs')

      "otherwise ..."

      #ASSIGN-VALUE! latest-return-point =
      #FIRST-ELEMENT-IN(return-point-list)

      #ASSIGN-VALUE! return-point-list =
      all-but-first-element-in (return-point-list)

      #RETURN-WITH-VALUE! latest-return-point

    #END #.

#DF all-but-first-element-in(list)

    "{list #EQ return-point-list}"

    => #TERMINAL-SUBSEQ-OF-LENGTH(#LENGTH(list) - 1) #OF
       list #.
```

```
#DF standard-name-of(name)

     "{ name #IS <numeric-variable> #U <string-variable> #U
     <variable> #U <simple-numeric-variable> #U
     <numeric-array-element> #U <control-variable> }"

     => standard-array-element-name-of(
        nameable-part-of(name)) #IF nameable-part-of(name)
        #IS <numeric-array-element> ;

     =>
        standard-parameter-name-derived-from(statement-containing
        (nameable-part-of(name))) #IF
        ($nameable-part-of(name)$)
        is-def-statement-parameter ;

     => #STRING-OF-TERMINALS-OF( nameable-part-of(name))
        #OTHERWISE #.


#DF standard-array-element-name-of(name)

     "{ name #IS <numeric-array-element> }"

     => one-dimension-array-element-name-of
        (numeric-array-name-of(name), first-dimension-value
        (subscript-part-of(name))
        ,"with-respect-to-the-bounds" option-base-for(name)
        ,"and" first-dimension-upper-bound-value-for(name))
        #IF ($subscript-part-of(name)$) has-one-dimension;

     => two-dimension-array-element-name-of
        (numeric-array-name-of(name), first-dimension-value
        (subscript-part-of(name)), second-dimension-value
        (subscript-part-of(name))
        ,"with-respect-to-the-bounds" option-base-for(name)
        ,"and" first-dimension-upper-bound-value-for(name)
        ,"and" second-dimension-upper-bound-value-for(name))
        #OTHERWISE #.


#DF one-dimension-array-element-name-of
(aname,index,base,bound)

     "{aname #IS <numeric-array-name> #AND index #IS
     #INTEGER #AND base #IS-IN \0,1\ #AND bound #IS
```

```
     #INTEGER}"

     => #STRING-OF-TERMINALS-OF(aname) #CW '(' #CW ($index$)
        with-leading-zeroes-suppressed #IF base <= index
        #AND index <= bound;

     => fatal-error('subscript out of bounds') #OTHERWISE #.


#DF two-dimension-array-element-name-of
(aname,idx1,idx2,base,bound1,bound2)

     "{aname #IS <numeric-array-name> #AND idx1 #IS #INTEGER
     #AND idx2 #IS #INTEGER #AND base #IS-IN \0,1\ #AND
     bound1 #IS #INTEGER #AND bound3 #IS #INTEGER}"

     => #STRING-OF-TERMINALS-OF(aname) #CW '(' #CW ($idx1$)
        with-leading-zeroes-suppressed #CW ',' #CW ($idx2$)
        with-leading-zeroes-suppressed #IF base <= idx1 #AND
        idx1 <= bound1 #AND base <= idx2 #AND idx2 <=
        bound2;

     => fatal-error('subscript out of bounds') #OTHERWISE #.


#DF first-dimension-value(sub)

     "{sub #IS <subscript>}"

     => ($($numeric-value (first-dimension-of(sub))$)
        rounded-to-an-integer$) converted-to-semanol-integer
        #.


#DF second-dimension-value(sub)

     "{sub #IS <subscript> #AND ($sub$) has-two-dimensions}"

     => ($($numeric-value (second-dimension-of(sub))$)
        rounded-to-an-integer$) converted-to-semanol-integer
        #.


#DF first-dimension-upper-bound-value-for(arrayel)

     "{arrayel #IS <numeric-array-element}"

     => #STRING-OF-TERMINALS-OF (first-dimension-bound-of
```

```
      (bounds-part-of (array-declaration-for
      (numeric-array-name-of(arrayel))))) #IF
      ($numeric-array-name-of(arrayel)$)
      is-explicitly-declared-array;

   => 10 #OTHERWISE #.


#DF second-dimension-upper-bound-value-for(arrayel)

   "{arrayel #IS <numeric-array-element}"

   => #STRING-OF-TERMINALS-OF (second-dimension-bound-of
      (bounds-part-of (array-declaration-for
      (numeric-array-name-of(arrayel))))) #IF
      ($numeric-array-name-of(arrayel)$)
      is-explicitly-declared-array;

   => 10 #OTHERWISE #.


#DF option-base-for(arrayel)

   "{ arrayel #IS <numeric-array-element> }"

   => '0' #IF sequence-of-option-statements-in
      (root-node(arrayel)) #EQ #NILSEQ ;

   => #STRING-OF-TERMINALS-OF (option-base-of
      (#FIRST-ELEMENT-IN (sequence-of-option-statements-in
      (root-node(arrayel))))) #OTHERWISE #.


#DF is-explicitly-declared-array(aname)

   "{ aname #IS <numeric-array-name> }"

   => #TRUE #IFF #THERE-EXISTS a #IN
      sequence-of-array-declarations-in (root-node(aname))
      #SUCH-THAT (aname #EQW numeric-array-name-of(a)) #.


#DF array-declaration-for(aname)

   "{ aname #IS <numeric-array-name> }"

   => #FIRST a #IN sequence-of-array-declarations-in
      (root-node(aname)) #SUCH-THAT (aname #EQW
```

```
     numeric-array-name-of(a)) #.


#DF is-def-statement-parameter(name)

     "{ name #IS <simple-numeric-variable> #U
     <string-variable> }"

     => ($ statement-containing(name), "has" name $)
        as-a-parameter #IF ($ statement-containing(name) $)
        is-def-statement-with-parameter ;

     => #FALSE #OTHERWISE #.


#DF as-a-parameter(def-st, "has" name)

     "{name #IS <string-variable> #U
     <simple-numeric-variable> &
       ($ def-st $) is-def-statement-with-parameter}"

     => #TRUE #IFF #STRING-OF-TERMINALS-OF(name) #EQW
        #STRING-OF-TERMINALS-OF(
        def-statement-parameter-of(def-st)) #.


#DF statement-containing(nx)

     "{nx #IS #NODE}"

     => statement-part-of(line-containing(nx)) #.


#DF standard-parameter-name-derived-from (def)

     "{ ($def$)is-def-statement-with-parameter }"

     => #STRING-OF-TERMINALS-OF( def-statement-name-of(def))
        #CW #STRING-OF-TERMINALS-OF(
        def-statement-parameter-of(def)) #.
```

```
#DF results-in-string-overflow(s)

      "{ s #IS #STRING}"

      => #TRUE #IFF #LENGTH(s) > max-assignable-string-length
         + 2 & first-character-in(s) #EQW '"' &
         last-character-in(s) #EQW '"' #OR #LENGTH(s) >
         max-assignable-string-length #.


#DF string-value(exp)

      "{ exp #IS <string-expression }"

      => #LATEST-VALUE(standard-name-of(
         string-variable-of(exp))) #IF
         ($exp$)is-string-variable ;

      => remove-quotes-from(#STRING-OF-TERMINALS-OF
         (string-constant-of(exp))) #IF ($exp$)
         is-string-constant #.


#DF remove-quotes-from(s)

      "{ s #IS #STRING }"

      => #SUBSTRING-OF-CHARACTERS 2 #TO #LENGTH(s) - 1 #OF s
         #.


#DF numeric-value(exp)

      "{ ($exp$) is-numeric-exp-subnode}"

      => numeric-value(operand-1-of(exp)) #IF exp #IS
         <numeric-expression> #U <term> #U <factor> #U
         <primary> ;

      => numeric-value(operand-1-of(exp)) #IF exp #IS
         <positive-expression> ;

      => perform( numeric-value(operand-1-of(exp)),
         'unary-minus', #UNDEFINED) #IF exp #IS <negation> ;

      => perform( numeric-value( operand-1-of(exp)), '+',
```

```
    numeric-value(operand-2-of(exp))) #IF exp #IS <sum>
    ;

 => perform( numeric-value( operand-1-of(exp)), '-',
    numeric-value(operand-2-of(exp))) #IF exp #IS
    <difference> ;

 => perform( numeric-value( operand-1-of(exp)), '*',
    numeric-value(operand-2-of(exp))) #IF exp #IS
    <product> ;

 => perform( numeric-value( operand-1-of(exp)), '/',
    numeric-value(operand-2-of(exp))) #IF exp #IS
    <quotient> ;

 => perform( numeric-value( operand-1-of(exp)), '^',
    numeric-value(operand-2-of(exp))) #IF exp #IS
    <involution> ;

 => #LATEST-VALUE(standard-name-of(exp)) #IF exp #IS
    <numeric-variable> ;

 => numeric-constant-value(exp) #IF exp #IS
    <numeric-rep> ;

 => numeric-function-value(exp) #IF exp #IS
    <numeric-function-ref> #.
```

"The BASIC standard prescribes certain actions for error conditions which can occur in the evaluation of the arithmetic operators. Overflow, division by zero and some special cases in involution are required to give a non-fatal error and return some standard result, possibly depending on one or both arguments. Underflow is not considered a non-fatal error, at least to the extent of listing it in the error section under the evaluation section. However, the remarks suggest that underflow be treated as an error, though clearly not requiring treatment as an error. We choose to model this by an implementation-dependent parameter whose value is #TRUE or #FALSE. If it is #TRUE, then underflow is uniformly treated as a non-fatal error, otherwise it returns the prescribed value, implementation-zero, with no other effect.

The BASIC standard specifies that non-fatal errors

shall be reported and subjected to the specified error recovery procedures. The text of the report is implementation dependent. Generally, the recovery procedure returns some specified value. Usually it returns one of the implementation dependent limits, such as implementation-infinity or implementation-zero. For many cases, the standard specifies that either plus or minus infinity is the non-fatal recovery procedure result. The sign of the result is determined from the operands."

```
#DF perform(op1,op,op2)

    "{ ($op1$) is-implementation-number & ($op2$)
    is-implementation-number & op #IS-IN
    \'unary-minus','+','-','*','/','^'\ }"

    => special-effect(op1,op,op2) #IF ($op1,op,op2$)
       requires-special-effect;

    => overflow-error-effect(op1,op,op2) #IF ($op1, op,
       op2$) results-in-overflow;

    => underflow-effect(op) #IF ($op1, op, op2$)
       results-in-underflow;

    => simple-perform(op1,op,op2) #OTHERWISE #.

#DF results-in-overflow(op1,op,op2)

    "{($op1$) is-implementation-number & ($op2$)
    is-implementation-number & op #IS-IN
    \'unary-minus','+','-','*','/','^'\}"

    => ($op1$) results-in-negate-overflow #IF op #EQW
       'unary-minus';

    => ($op1,op2$) results-in-add-overflow #IF op #EQW '+';

    => ($op1,op2$) results-in-subtract-overflow #IF op #EQW
       '-';

    => ($op1,op2$) results-in-multiply-overflow #IF op #EQW
       '*';

    => ($op1,op2$) results-in-divide-overflow #IF op #EQW
       '/';
```

```
     => ($op1,op2$) results-in-involute-overflow #IF op #EQW
        '^' #.

#PROC-DF overflow-error-effect(op1,op,op2)

     "{($op1$) is-implementation-number & ($op2$)
     is-implementation-number & op #IS-IN
     \'unary-minus','+','-','*','/','^'\}"

     #BEGIN

       #COMPUTE! non-fatal-overflow-error-report(op)

       #IF overflow-result-sign(op1,op,op2) #EQW '+'
       #THEN #RETURN-WITH-VALUE! implementation-infinity

       #IF overflow-result-sign(op1,op,op2) #EQW '-'
       #THEN #RETURN-WITH-VALUE!
       implementation-negative-infinity

     #END #.

#DF results-in-underflow(op1,op,op2)

     "{($op1$) is-implementation-number & ($op2$)
     is-implementation-number & op #IS-IN
     \'unary-minus','+','-','*','/','^'\}"

     => ($op1$) results-in-negate-underflow #IF op #EQW
        'unary-minus';

     => ($op1,op2$) results-in-add-underflow #IF op #EQW
        '+';

     => ($op1,op2$) results-in-subtract-underflow #IF op
        #EQW '-';

     => ($op1,op2$) results-in-multiply-underflow #IF op
        #EQW '*';

     => ($op1,op2$) results-in-divide-underflow #IF op #EQW
        '/';

     => ($op1,op2$) results-in-involute-underflow #IF op
        #EQW '^' #.
```

```
#PROC-DF underflow-effect(op)

     "{op #IS-IN \'unary-minus','+','-','*','/','^'\}"

     #BEGIN

       #IF underflow-is-a-detected-non-fatal-error
       #THEN #COMPUTE! non-fatal-underflow-error-report(op)

       #RETURN-WITH-VALUE! implementation-zero

     #END #.


#DF non-fatal-overflow-error-report(op)

     "{op #IS-IN \'unary-minus','+','-','*','/','^'\}"

     => non-fatal-negate-overflow-error-report #IF op #EQW
        'unary-minus';

     => non-fatal-add-overflow-error-report #IF op #EQW '+';

     => non-fatal-subtract-overflow-error-report #IF op #EQW
        '-';

     => non-fatal-multiply-overflow-error-report #IF op #EQW
        '*';

     => non-fatal-divide-overflow-error-report #IF op #EQW
        '/';

     => non-fatal-involute-overflow-error-report #IF op #EQW
        '^' #.


#DF overflow-result-sign(op1,op,op2)

     "{($op1$) is-implementation-number & ($op2$)
     is-implementation-number & op #IS-IN
     \'unary-minus','+','-','*','/','^'\}"

     => negate-overflow-result-sign(op1) #IF op #EQW
        'unary-minus';

     => add-overflow-result-sign(op1,op2) #IF op #EQW '+';
```

======================= control-59 =======================


```
    => subtract-overflow-result-sign(op1,op2) #IF op #EQW
       '-';

    => multiply-overflow-result-sign(op1,op2) #IF op #EQW
       '*';

    => divide-overflow-result-sign(op1,op2) #IF op #EQW
       '/';

    => involute-overflow-result-sign(op1,op2) #IF op #EQW
       '^' #.

#DF non-fatal-underflow-error-report(op)

    "{op #IS-IN \'unary-minus','+','-','*','/','^'\}"

    => non-fatal-negate-underflow-error-report #IF op #EQW
       'unary-minus';

    => non-fatal-add-underflow-error-report #IF op #EQW
       '+';

    => non-fatal-subtract-underflow-error-report #IF op
       #EQW '-';

    => non-fatal-multiply-underflow-error-report #IF op
       #EQW '*';

    => non-fatal-divide-underflow-error-report #IF op #EQW
       '/';

    => non-fatal-involute-underflow-error-report #IF op
       #EQW '^' #.

#DF simple-perform(op1,op,op2)

    "{($op1$) is-implementation-number & ($op2$)
    is-implementation-number & op #IS-IN
    \'unary-minus','+','-','*','/','^'\}"

    => implementation-negate(op1) #IF op #EQW
       'unary-minus';

    => implementation-add(op1,op2) #IF op #EQW '+';
```

====================== control-60 ======================


```
     => implementation-subtract(op1,op2) #IF op #EQW '-';

     => implementation-multiply(op1,op2) #IF op #EQW '*';

     => implementation-divide(op1,op2) #IF op #EQW '/';

     => implementation-involute(op1,op2) #IF op #EQW '^' #.


#DF requires-special-effect(op1,op,op2)

     "{($op1$) is-implementation-number & ($op2$)
     is-implementation-number & op #IS-IN
     \'unary-minus','+','-','*','/','^'\}"

     => #FALSE #IF op #IS-IN \'unary-minus','+','-','*'\;

     => ($op1,op2$) requires-special-divide-effect #IF op
        #EQW '/';

     => ($op1,op2$) requires-special-involute-effect #IF op
        #EQW '^' #.


#DF special-effect(op1,op,op2)

     "{($op1$) is-implementation-number & ($op2$)
     is-implementation-number & op #IS-IN \'/','^'\}"

     => special-divide-effect(op1) #IF op #EQW '/';

     => special-involute-effect(op1,op2) #IF op #EQW '^' #.


"Divide has one special case, division by zero."


#DF requires-special-divide-effect(op1,op2)

     "{ ($op1$) is-implementation-number & ($op2$)
     is-implementation-number}"

     => #TRUE #IFF implementation-equals-test (op2,
        implementation-zero) #.


#PROC-DF special-divide-effect(op1)
```

```
     "{($op1$) is-implementation-number & ($op2$)
     is-implementation-number}"

     #BEGIN

       #COMPUTE! non-fatal-divide-by-zero-error-report

       #IF divide-by-zero-result-sign(op1) #EQW '+'
       #THEN #RETURN-WITH-VALUE! implementation-infinity

       #IF divide-by-zero-result-sign(op1) #EQW '-'
       #THEN #RETURN-WITH-VALUE!
       implementation-negative-infinity

     #END #.
```

"Involution has three special cases. They are 0^0, 0^(negative), and (negative)^(non-integer)."

```
#DF requires-special-involute-effect(op1,op2)

     "{($op1$) is-implementation-number & ($op2$)
     is-implementation-number}"

     => #TRUE #IF implementation-equals-test (op1,
        implementation-zero) & implementation-equals-test
        (op2, implementation-zero);

     => #TRUE #IF implementation-equals-test (op1,
        implementation-zero) & implementation-less-than-test
        (op2, implementation-zero);

     => #TRUE #IF implementation-less-than-test (op1,
        implementation-zero) & #NOT ($op2$)
        is-implementation-integer;

     => #FALSE #OTHERWISE  #.


#PROC-DF special-involute-effect(op1,op2)

     "{($op1$) is-implementation-number & ($op2$)
     is-implementation-number}"

     #BEGIN
```

```
    #IF implementation-equals-test (op1,
    implementation-zero)
    #THEN

    #BEGIN

      #IF implementation-equals-test (op2,
      implementation-zero)
      #THEN

      "Case of zero^zero"

      #RETURN-WITH-VALUE! implementation-one

      "Case of zero to negative power."

      #COMPUTE!
      non-fatal-zero-involuted-to-negative-error-report

      #RETURN-WITH-VALUE! implementation-infinity

    #END

    "the only case left by this point in this df is a
    negative involuted to a non-integer power."

    #COMPUTE! fatal-error('negative involuted to
    non-integer')

  #END #.



#DF numeric-function-value(ref)

    "{ ref #IS <numeric-function-ref> }"

    => numeric-defined-function-value
       (numeric-defined-function-ref-of(ref)) #IF ($ref$)
       is-numeric-defined-function-ref;

    => numeric-supplied-function-value
       (numeric-supplied-function-ref-of(ref)) #OTHERWISE
       #.
```

====================== control-63 =====================


```
#PROC-DF numeric-defined-function-value(dref)

      "{ dref #IS <numeric-defined-function-ref> }"

      #BEGIN

        #IF ($dref$) has-an-argument
        #THEN #COMPUTE! #ASSIGN-LATEST-VALUE
        (standard-parameter-name-derived-from
        (def-statement-with-name
        (numeric-defined-function-name-of(dref))) ,"the
        value" argument-value-of(dref))

        #RETURN-WITH-VALUE! numeric-value
        (def-statement-expression-of (def-statement-with-name
        (numeric-defined-function-name-of(dref))))

      #END #.


#DF def-statement-with-name(dname)

      "{dname #IS <numeric-defined-function>}"

      => #FIRST x #IN (#SEQUENCE-OF <def-statement> #IN
         root-node(dname)) #SUCH-THAT(
         #STRING-OF-TERMINALS-OF( def-statement-name-of(x))
         #EQW #STRING-OF-TERMINALS-OF(dname)) #.


#DF argument-value-of(ref)

      "{ref #IS <numeric-defined-function-ref> #U
      <numeric-supplied-function-ref> #AND ($ref$)
      has-an-argument}"

      => numeric-value (argument-expression-of(ref)) #.


#DF numeric-supplied-function-value(sref)

      "{sref #IS <numeric-supplied-function-ref>}"

      => abs-function-value(argument-value-of(sref)) #IF
         ($sref$) is-abs-function-ref;

      => atn-function-value(argument-value-of(sref)) #IF
         ($sref$) is-atn-function-ref;
```

```
     => cos-function-value(argument-value-of(sref)) #IF
        ($sref$) is-cos-function-ref;

     => exp-function-value(argument-value-of(sref)) #IF
        ($sref$) is-exp-function-ref;

     => int-function-value(argument-value-of(sref)) #IF
        ($sref$) is-int-function-ref;

     => log-function-value(argument-value-of(sref)) #IF
        ($sref$) is-log-function-ref;

     => rnd-function-value #IF ($sref$) is-rnd-function-ref;

     => sgn-function-value(argument-value-of(sref)) #IF
        ($sref$) is-sgn-function-ref;

     => sin-function-value(argument-value-of(sref)) #IF
        ($sref$) is-sin-function-ref;

     => sqr-function-value(argument-value-of(sref)) #IF
        ($sref$) is-sqr-function-ref;

     => tan-function-value(argument-value-of(sref)) #IF
        ($sref$) is-tan-function-ref #.


#DF abs-function-value(n)

     "{($n$) is-implementation-number}"

     => implementation-negate(n) #IF
        implementation-less-than-test
        (n,implementation-zero) ;

     => n #OTHERWISE #.


#DF atn-function-value(n)

     "{($n$) is-implementation-number}"

     => implementation-arctangent-function(n) #.


#DF cos-function-value(n)
```

```
     "{($n$) is-implementation-number}"

     => implementation-cosine-function(n) #.


#DF exp-function-value(n)

     "{($n$) is-implementation-number}"

     => exponential-function-overflow-effect(n) #IF ($n$)
        results-in-exponential-function-overflow;

     => exponential-function-underflow-effect(n) #IF ($n$)
        results-in-exponential-function-underflow;

     => implementation-exponential-function(n) #OTHERWISE #.


#DF int-function-value(n)

     "{($n$) is-implementation-number}"

     => implementation-integer-function(n) #.


#DF log-function-value(n)

     "{($n$) is-implementation-number}"

     => special-logarithm-function-effect(n) #IF #NOT
        implementation-greater-than-test
        (n,implementation-zero);

     => implementation-logarithm-function(n) #OTHERWISE #.


"The rnd function is peculiar.  It has no direct argument,
yet it is affected by the occurrence of a randomize
statement in a program.  Therefore, the presence of the
randomize stement is passed as an argument to the
implementation dependent section.  The argument is a boolean
which is #TRUE if the randomize statement is present."


#DF rnd-function-value

     => implementation-random-function(#TRUE) #IF
        randomize-occurs-in-program;
```

```
     => implementation-random-function(#FALSE) #OTHERWISE #.

#DF randomize-occurs-in-program

     => #TRUE #IFF  #THERE-EXISTS x #IN
        (#SEQUENCE-OF-NODES-IN(basic-program)) #SUCH-THAT( x
        #IS <randomize-statement>) #.

#DF sgn-function-value(n)

     "{($n$) is-implementation-number}"

     => implementation-negative-one #IF
        implementation-less-than-test
        (n,implementation-zero);

     => implementation-zero #IF implementation-equals-test
        (n,implementation-zero);

     => implementation-one #OTHERWISE #.

#DF sin-function-value(n)

     "{($n$) is-implementation-number}"

     => implementation-sine-function(n) #.

#DF sqr-function-value(n)

     "{($n$) is-implementation-number}"

     => special-square-root-function-result(n) #IF
        implementation-less-than-test
        (n,implementation-zero);

     => implementation-square-root-function(n) #OTHERWISE #.

#DF tan-function-value(n)

     "{($n$) is-implementation-number}"

     => tangent-function-overflow-effect(n) #IF ($n$)
```

```
    results-in-tangent-function-overflow;

  => implementation-tangent-function(n) #OTHERWISE #.

#PROC-DF special-logarithm-function-effect(n)

    "{($n$) is-implementation-number}"

    #BEGIN

      #COMPUTE!  fatal-error('non-positive argument to LOG
      function')

    #END #.

#PROC-DF special-square-root-function-result(n)

    "{($n$) is-implementation-number}"

    #BEGIN

      #COMPUTE! fatal-error('negative argument to SQR
      function')

    #END #.

#PROC-DF exponential-function-overflow-effect(n)

    "{($n$) is-implementation-number}"

    #BEGIN

      #COMPUTE!
      non-fatal-exponential-function-overflow-error-report

      #IF exponential-function-result-sign (n) #EQW '+'
      #THEN #RETURN-WITH-VALUE! implementation-infinity

      #IF exponential-function-result-sign (n) #EQW '-'
      #THEN #RETURN-WITH-VALUE!
      implementation-negative-infinity

    #END #.
```

```
#PROC-DF tangent-function-overflow-effect(n)

     "{($n$) is-implementation-number}"

     #BEGIN

       #COMPUTE!
       non-fatal-tangent-function-overflow-error-report

       #IF tangent-function-result-sign (n) #EQW '+'
       #THEN #RETURN-WITH-VALUE! implementation-infinity

       #IF tangent-function-result-sign (n) #EQW '-'
       #THEN #RETURN-WITH-VALUE!
       implementation-negative-infinity

     #END #.

#PROC-DF exponential-function-underflow-effect(n)

     "{($n$) is-implementation-number}"

     #BEGIN

       #COMPUTE!
       non-fatal-exponential-function-underflow-error-report

       #RETURN-WITH-VALUE! implementation-zero

     #END #.
```

"The following definition of numeric-constant-value utilizes the same implementation-dependent functions as input for conversion and error testing. The BASIC standard does not state that input conversion is the same as numeric constant conversion in programs, but it seems to be a reasonable assumption."

```
#DF numeric-constant-value(n)

     "{n #IS <numeric-rep> #U <numeric-constant> }"

     => numeric-constant-overflow-error-effect
        (#STRING-OF-TERMINALS-OF(n)) #IF
        ($#STRING-OF-TERMINALS-OF (n)$)
```

```
        results-in-numeric-conversion-overflow;

     => numeric-constant-underflow-effect #IF
        ($#STRING-OF-TERMINALS-OF(n)$)
        results-in-numeric-conversion-underflow;

     => implementation-numeric-representation
        (#STRING-OF-TERMINALS-OF(n)) #OTHERWISE #.


#PROC-DF numeric-constant-overflow-error-effect(s)

     "{s #IS #STRING &
     #CONTEXT-FREE-PARSE-TREE(s,<numeric-constant>) #IS
     #NODE}"

     #BEGIN

       #COMPUTE!
       non-fatal-numeric-constant-overflow-error-report

       #IF numeric-constant-overflow-result-sign(s) #EQW '+'
       #THEN

       #RETURN-WITH-VALUE! implementation-infinity

       #IF numeric-constant-overflow-result-sign(s) #EQW '-'
       #THEN

       #RETURN-WITH-VALUE! implementation-negative-infinity

     #END #.


#PROC-DF numeric-constant-underflow-effect

   #BEGIN

     #COMPUTE!
     non-fatal-numeric-constant-underflow-error-report

     #RETURN-WITH-VALUE! implementation-zero

   #END #.


#DF relation-value (rel-exp)
```

```
     "{rel-exp #IS <relational-expression> }"

     => string-relation-value (rel-exp) #IF ($rel-exp$)
        is-string-relational-expression;

     => numeric-relation-value (rel-exp) #IF ($rel-exp$)
        is-numeric-relational-expression #.


#DF string-relation-value (rel-exp)

     "{($rel-exp$) is-string-relational-expression}"

     => apply-string-relation-test (string-value
        (operand-1-of (rel-exp)), relation-of (rel-exp),
        string-value (operand-2-of (rel-exp))) #.


#DF apply-string-relation-test (opd1, relop, opd2)

     "{opd1 #IS #STRING & relop #IS <equality-relation> &
     opd2 #IS #STRING}"

     => string-equals-test (opd1,opd2) #IF relop #EQW '=';

     => string-not-equals-test (opd1, opd2) #IF relop #EQW
        '<>' #.


#DF string-equals-test (opd1,opd2)

     "{ opd1 #IS #STRING & opd2 #IS #STRING}"

     => opd1 #EQW opd2 #.


#DF string-not-equals-test (opd1, opd2)

     "{opd1 #IS #STRING & opd2 #IS #STRING}"

     => opd1 #NEQW opd2 #.

     "COMMENT: The BASIC standard does not specify enough
     explicitly about the nature of the numeric relationals.
     In particular, it is not stated whether any error
     conditions are associated with the relations.  If the
     implementation were to use subtraction followed by a
     comparison to zero as the basis for defining the
```

relations, then the subtraction could result in overflow. Since the standard does not explicitly say that overflow can occur, it must be assumed that it cannot and so the implementor must guard against any errors occurring in numeric relations."

```
#DF numeric-relation-value (rel-exp)

     "{($rel-exp$)is-relational-expression}"

     => apply-numeric-relation-test (numeric-value
        (operand-1-of (rel-exp)), relation-of (rel-exp),
        numeric-value (operand-2-of (rel-exp))) #.


#DF apply-numeric-relation-test (opd1, relop, opd2)

     "{($opd1$) is-implementation-number & relop #IS
     <relation> & ($opd2$) is-implementation-number}"

     => implementation-equals-test (opd1,opd2) #IF relop
        #EQW '=';

     => implementation-not-equals-test (opd1, opd2) #IF
        relop #EQW '<>';

     => implementation-less-than-test (opd1, opd2) #IF relop
        #EQW '<';

     => implementation-greater-than-test (opd1, opd2) #IF
        relop #EQW '>';

     => implementation-not-less-test (opd1, opd2) #IF relop
        #EQW '>=';

     => implementation-not-greater-test (opd1, opd2) #IF
        relop #EQW '<='#.
```

"In a number of places in the specification of BASIC, certain numbers are required to be integers, such as the TAB function or array bounds. To describe these things, it is convenient to convert the implementation numbers to semanol integers and operate on these integers. This can be accomplished in an implementation independent fashion by first converting the implementation dependent number to a canonical standard form number and then converting that to a SEMANOL integer."

```
#DF converted-to-semanol-integer(r)

     "{ ($r$) is-implementation-integer }"

     => convert-canonical-float-to-semanol-integer (($r$)
        converted-to-canonical-float) #.

#DF convert-canonical-float-to-semanol-integer(n)

     "{($n$) is-canonical-float}"

     => significand-part(n) #CW ($exrad-part(n)$) zeros #.

     "Define a conversion function to convert an
     implementation-number to a cononical form basic
     constant."

#DF converted-to-canonical-float (r)

     "{($r$) is-implementation-number}"

     => ($($r$) converted-to-standard-float$)
        in-canonical-form #.
```

"In order to define the proper output form of numbers, the implementor is required to define two parameters, the implementation-significance-width, also represented as the letter d, and the implementation-exrad-width, or e. The standard does not seem to require that d or e have any relation to the implementation precision or range. There is

a requirement that d must be at least 6 and e must be at least 2. The resolved questions on page 35 are not part of the standard. It contains the only guideline for choosing d and e. The statement there is that the implementor may choose d and e to allow output of all numeric representations in the range of the implementation.

For the purpose of determining the output format for a given number, the class of possible numbers is divided into four classes.

a. integers whose magnitude is in the range 0 to (but not including) 10^d.

b. non-integers whose magnitude is in the range 0.1 - 0.5*10^(-d-1) to (but not including) 10^d - 0.5 .

c. Numbers less than 0.1 - 0.5*10^(-d-1) which can be represented exactly in d decimal digits.

d. all other numbers.

This classification is derived from page 33, lines 1 thru 30. The possible formats are defined on page 13,lines 12 thru 15. Class a numbers are output using the so-called NR1 format. Class b use NR2. Class c also uses NR2 format by virtue of the fact that all its members can be exactly represented in the NR2 form. Note that for non-decimal implementations, numbers which are potentialy in class c must be converted to decimal representation on a trial basis. Class d uses NR3 format.

The class b limits require some interpretation. We take it that the intent of class b is to describe those implementation numbers in the range of 0.1 upto 10^d. The subtractive factors of one half and one half times 10^(-d-1) would seem to represent the idea that the numbers to be included are those which can be rounded in the d+1 th significant digit to produce a number in the range 0.1 to 10^d with d or fewer significant digits.

The function to convert a number to output format is implementation independent. This is possible because the implementation number to be output is first converted to a BASIC constant. This BASIC constant can then be transformed to the appropriate output form using the standard arithmetic. Note that the following function converts the implementation number to a canonical basic constant. Subsequent functions are defined to operate on canonical numeric constants to simplify the definition."

```
#DF numeric-output-representation(n)

     "{($n$) is-implementation-number}"
```

```
     => canonical-number-output-representation (($n$)
        converted-to-canonical-float) #.


#DF canonical-number-output-representation (n)

     "{($n$) is-canonical-float}"

     => ($n$) in-output-class-a-format #IF ($n$)
        is-in-output-class-a;

     => ($($n$) rounded-for-significance-width-output$)
        in-output-class-b-format #IF ($n$)
        is-in-output-class-b;

     => ($n$) in-output-class-c-format #IF ($n$)
        is-in-output-class-c;

     => ($n$) in-output-class-d-format #OTHERWISE #.


#DF output-sign-string(n)

     "{ ($n$) is-canonical-float }"

     => '-' #IF significand-part(n) < 0;

     => #SPACE #OTHERWISE #.

     "Class a numbers are integers with magnitudes between 0
     and 10^d.  They are output as sdd...dd (NR1 format)"


#DF is-in-output-class-a(n)

     "{ ($n$) is-canonical-float }"

     =>  #TRUE #IFF ($n$) is-canonical-integer & ($
        standard-abs(n) , output-class-a-maximum$)
        is-standard-less-than #.


#DF in-output-class-a-format(n)

     "{ ($n$) is-canonical-float & ($n$)
     is-in-output-class-a }"
```

```
     => output-sign-string(n) #CW #ABS(significand-part(n))
        #CW ($exrad-part(n)$) zeros #CW #SPACE #.
```

"Class b numbers are non-integers with magnitudes between (approximately) 0.1 and 10^d. They are output as sdd...d.dd...d (NR2 format)"

```
#DF is-in-output-class-b(n)

     "{ ($n$) is-canonical-float }"

     => #TRUE #IFF ($standard-abs(n),
        output-class-b-maximum$) is-standard-less-than &
        #NOT ($standard-abs(n), output-class-b-minimum$)
        is-standard-less-than #.
```

"Class b rounding uses the p-th-digit rounding function defined for canonical-form numbers. The rounding takes place in the dth digit of the significand of the number."

```
#DF rounded-for-significance-width-output(n)

     "{($n$) is-canonical-float}"

     => ($n ,"and-a-p-of"
        implementation-significance-width$)
        rounded-to-p-digits #.


#DF in-output-class-b-format(n)

     "{ ($n$) is-canonical-float & ($n$)
     is-in-output-class-b & #LENGTH(significand-part(n)) <=
     implementation-significance-width }"

     => output-sign-string(n) #CW class-b-significand
        (#ABS(significand-part(n)) ,"with-respect-to"
        exrad-part(n)) #CW #SPACE #.


#DF class-b-significand(s,e)

     "{($construct-float(s,e)$) is-canonical-float & s > 0 &
     ($construct-float(s,e)$) is-in-output-class-b &
```

```
     #LENGTH(s) <= implementation-significance-width}"

     => s #CW ($e$) zeros #CW '.' #IF e >= 0;

     => (#LEFT (#LENGTH(s) + e) #CHARACTERS-OF s) #CW '.'
        #CW (#RIGHT (#NEG e) #CHARACTERS-OF s) #IF
        #LENGTH(s) >= #NEG e;

     => '.' #CW significand-part(s) #OTHERWISE #.

     "Class c numbers are numbers less than 0.1 that are
     exactly representable in d digits.  They are output as
     s.dd...d (NR2 format)"


#DF is-in-output-class-c(n)

     "{($n$) is-canonical-float}"

     => #TRUE #IFF ($standard-abs(n),
        output-class-c-maximum$) is-standard-less-than &
        ($n$) is-exactly-representable-for-class-c #.


#DF is-exactly-representable-for-class-c(n)

     "{($n$) is-canonical-float &
     ($standard-abs(n),implementation-one$)
     is-standard-less-than}"

     => #TRUE #IFF #NEG exrad-part(n) <=
        implementation-significance-width #.


#DF in-output-class-c-format(n)

     "{($n$) is-canonical-float & ($n$)
     is-in-output-class-c}"

     => output-sign-string(n) #CW '.' #CW ($#NEG
        exrad-part(n) - #LENGTH(#ABS(significand-part(n)))$)
        zeros #CW #ABS(significand-part(n)) #CW #SPACE #.


"All other numbers fall into class d.  The standard
specifies a particular form of the NR3 format for this
output class.  It clearly states that the trailing zeros in
the significand are not omitted, but it does not state
```

whether leading zeros in the exrad may be included. This definition assumes that leading zeros are always suppressed in the exrad. The class d numbers are output as sdd...d.dd...ddEsee (NR3 format)"

```
#DF in-output-class-d-format(n)

     "{($n$) is-canonical-float}"

     => output-sign-string(n) #CW class-d-significand
        (#ABS(significand-part(n))) #CW 'E' #CW
        class-d-exrad (exrad-part(n) ,"with-respect-to"
        #ABS(significand-part(n))) #CW #SPACE #.


#DF class-d-significand(s)

     "{s #IS #INTEGER & s >= 0}"

     => first-character-in(s) #CW '.' #CW
        all-but-first-character-in(s) #CW
        ($implementation-significance-width - #LENGTH(s)$)
        zeros #IF #LENGTH(s) <
        implementation-significance-width;

     => first-character-in(s) #CW '.' #CW (#LEFT
        (implementation-significance-width - 1)
        #CHARACTERS-OF all-but-first-character-in(s))
        #OTHERWISE #.


#DF class-d-exrad(e ,"with-respect-to" s)

     "{e #IS #INTEGER & s #IS #INTEGER & s >= 0}"

     => exrad-output-sign(e + #LENGTH(s) - 1) #CW ($e +
        #LENGTH(s) - 1$) with-leading-zeroes-suppressed #.


#DF exrad-output-sign(e)

     "{e #IS #INTEGER}"

     => '-' #IF e < 0;

     => '+' #OTHERWISE #.
```

"The standard specifies the output classes in terms of two parameters, implementation-significance-width and the implementation-exrad-width. The latter width is defined in the standard but is never used. The standard uses the widths to define some maxima and minima for the various classes."

"The class a maximum must be 1Ed."

```
#DF output-class-a-maximum

     => construct-float
        (1,implementation-significance-width) #.
```

"The class b maximum must be 1Ed - 5E-1."

```
#DF output-class-b-maximum

     => construct-float
        (($implementation-significance-width$) nines #CW
        '5', -1) #.
```

"The class b minimum must be 1E-1 - 5E(-d-2)."

```
#DF output-class-b-minimum

     => construct-float
        (($implementation-significance-width$) nines #CW
        '5', #NEG implementation-significance-width - 2) #.
```

"The class c maximum must be the same as the class b minimum."

```
#DF output-class-c-maximum

     => output-class-b-minimum #.

#DF nines (n)

     "{n #IS #INTEGER #AND n >= 0}"
```

AD-A040 990 TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CALIF F/G 9/2

MINIMAL BASIC SEMANOL (76) SPECIFICATION LISTING.(U)

MAY 77 F C BELZ, R M HART, D M HEIMBIGNER F30602-76-C-0245

UNCLASSIFIED RADC-TR-77-170-VOL-2 NL

2 OF 2

ADA040990

END

DATE

FILMED

7-77

```
=> '9999999999' #CW ($n - 10$) nines #IF n > 10;

=> #LEFT n #CHARACTERS-OF '9999999999' #OTHERWISE #.
```

"A standard arithmetic is defined as a standard of comparison for the implementation dependent arithmetic. The standard arithmetic was defined to operate on BASIC numeric-constants (see the CONTEXT-FREE-SYNTAX section). Defining arithmetic on the full range of BASIC numeric-constants is difficult. Therefore, a two step approach is used. In the first step(performed by the DF in-canonical-form), an arbitrary BASIC numeric-constant is transformed into another BASIC constant of a special form. The arithmetic operators then calculate their results from numbers in this canonical form. An examination of the standard-add DF shows how this two step process works in detail. The canonical form of BASIC numeric-constants are defined by the following grammar and equivalently by the DF is-canonical-float.

```
      #DF canonical-numeric-constant
            => <#NILSET #U
<'-'>><standard-significand><'E'>
                  <#NILSET #U <'-'>><standard-exrad> #.
      #DF standard-significand
            => %1<'0'>
            => <%<#DIGIT>><#DIGIT #S- <'0'>> #.
      #DF standard-exrad
            => %1<#DIGIT> #.
```

The value of the canonical constant aEb is a*10^b."

```
#DF is-standard-float(n)

     "{n #IS #STRING}"

     => #TRUE #IFF n #IS <numeric-constant> #.


#DF is-canonical-float(r)

     "{r #IS <numeric-constant> }"

     => #TRUE #IFF significand-part(r) #IS #INTEGER &
        exrad-part(r) #IS #INTEGER & (last-character-in
        (significand-part(r)) #NEQW '0' #OR
        significand-part(r) = 0) #.
```

"Converting an arbitrary BASIC constant into canonical form

involves four sub-steps. These steps are tested and applied one at a time. The steps are:

1. Append a 'E0' to the constant if it does not already have an exrad.
2. Remove any leading plus signs in the exrad or significand.
3. Remove the decimal point from the significand. This may require adjusting the exrad to leave the value of the constant unchanged.
4. Remove any trailing zeros in the significand. Again, adjustment of the exrad may be necessary."

```
#DF in-canonical-form(r)

      "{r #IS <numeric-constant> }"

      => ($($r$) with-exrad-appended$) in-canonical-form #IF
         exrad-part(r) #IS #UNDEFINED ;

      => ($($ significand-part(r), exrad-part(r) $)
         with-leading-plus-signs-removed$) in-canonical-form
         #IF first-character-in (significand-part(r)) #EQW
         '+' #OR first-character-in (exrad-part(r)) #EQW '+';

      => ($($ significand-part(r), exrad-part(r) $)
         with-decimal-point-removed$) in-canonical-form #IF
         '.' #IS #SUBWORD significand-part(r);

      => ($ significand-part(r), exrad-part(r) $)
         with-trailing-zeros-removed #IF significand-part(r)
         #N= '0' & last-character-in (significand-part(r))
         #EQW '0';

      => r #OTHERWISE #.


#DF with-exrad-appended(r)

      "{r #IS <numeric-constant> & exrad-part(r) #IS
      #UNDEFINED}"

      => r #CW 'E0' #.


#DF with-leading-plus-signs-removed(m,e)
```

```
     "{m #IS significand & e #IS exrad &
     first-character-in(m) #EQW '+' #OR
     first-character-in(e) #EQW '+'}"

     => construct-float( all-but-first-character-in(m),
        all-but-first-character-in(e)) #IF
        first-character-in(m) #EQW '+' &
        first-character-in(e) #EQW '+';

     => construct-float( all-but-first-character-in(m), e)
        #IF first-character-in(m) #EQW '+';

     => construct-float(m, all-but-first-character-in(e))
        #OTHERWISE #.


#DF all-but-first-character-in(s)

     "{s #IS #STRING & #LENGTH(s) >= 1}"

     => #RIGHT #LENGTH(s) - 1 #CHARACTERS-OF s #.


#DF with-decimal-point-removed(m,e)

     "{m #IS significand & e #IS exrad & '.' #IS #SUBWORD m
     & e #IS #INTEGER}"

     => construct-float ((#PREFIX-OF-FIRST '.' #IN m) #CW
        (#SUFFIX-OF-FIRST '.' #IN m), e - #LENGTH
        (#SUFFIX-OF-FIRST '.' #IN m)) #.


#DF with-trailing-zeros-removed(m,e)

     "{m #IS significand & e #IS exrad & last-character-in
     (m) #EQW '0' & m #IS #INTEGER & m #N= 0 & e #IS
     #INTEGER}"

     => construct-float (($m$) without-trailing-zeros, e +
        #LENGTH(m) - #LENGTH(($m$) without-trailing-zeros))
        #.


#DF without-trailing-zeros(n)

     "{n #IS #INTEGER  & n #N= 0 & last-character-in(n) #EQW
     '0'}"
```

```
     => #LEFT index-of-last-non-zero-in(n) #CHARACTERS-OF
        n#.

#DF index-of-last-non-zero-in(n)

     "{ n #IS #INTEGER & n #N= 0 & last-character-in(n) #EQW
     '0'}"

     => #LAST i : 1 <= i <= #LENGTH(n) #SUCH-THAT (i
        #TH-CHARACTER-IN n #NEQW '0') #.

"Define the selector and constructor functions for floating
point constants."

#DF significand-part(r)

     "{ r #IS <numeric-constant> }"

     => #PREFIX-OF-FIRST 'E' #IN r #.

#DF exrad-part(r)

     "{r #IS <numeric-constant> }"

     => #SUFFIX-OF-FIRST 'E' #IN r #.

#DF construct-float(m,e)

     "{m #IS significand & e #IS exrad}"

     => m #CW 'E' #CW e #.

     "Define a predicate on the canonical form numbers which
     is true IFF the number represents an integer.  Since a
     canonical integer has a significand which has its radix
     point at the right, a canonical number will be an
     integer IFF its exrad is non-negative."

#DF is-canonical-integer(n)

     "{($n$) is-canonical-float}"
```

```
     => #TRUE #IFF exrad-part(n) >= 0 #.

     "Define a rounding function on canonical form numbers.
     The rounding takes place on the pth digit of the
     significand of the number(possibly extended with zeros
     to at least p digits).  The result is a canonical
     number with p or fewer digits in the significand."


#DF rounded-to-p-digits(n,p)

     "{($n$) is-canonical-float & p #IS #INTEGER & p > 0}"

     => ($($canonical-add(n, rounding-factor-for(n,p)), p$)
        truncated-after-the-p-th-digit$) in-canonical-form
        #.


#DF rounding-factor-for(n,p)

     "{($n$) is-canonical-float & p #IS #INTEGER & p > 0}"

     => construct-float (sign-string(significand-part(n))
        #CW 5, exrad-part(n) +
        #LENGTH(#ABS(significand-part(n))) - p - 1) #.


#DF truncated-after-the-p-th-digit(n,p)

     "{($n$) is-canonical-float & p #IS #INTEGER & p > 0}"

     => n #IF #LENGTH(#ABS(significand-part(n))) <= p;

     => precision-limited (significand-part(n),
        exrad-part(n), p) #OTHERWISE #.


"Define two elementary numeric functions on floating point
numbers, SIGN and ABS functions."


#DF standard-sign(r)

     "{r #IS <numeric-constant> }"

     => canonical-sign (($r$) in-canonical-form) #.
```

```
#DF canonical-sign(r)

     "{($r$) is-canonical-float}"

     => #SIGN(significand-part(r)) #.


#DF standard-abs(r)

     "{ r #IS <numeric-constant> }"

     => canonical-abs (($r$) in-canonical-form) #.


#DF canonical-abs(r)

     "{($r$) is-canonical-float}"

     => construct-float (#ABS(significand-part(r)),
        exrad-part(r)) #.

     "Floating point addition is defined, if the exrads are
     equal, as the canonical result of adding the
     significands to get the result significand and using
     the common exrad.
         If the two exrads are unequal, the number with the
     larger exrad is aligned to the smaller exrad and then
     added as described."


#DF standard-add(rx,ry)

     "{rx #IS <numeric-constant> & ry #IS <numeric-constant>
     }"

     => canonical-add (($rx$) in-canonical-form, ($ry$)
        in-canonical-form) #.


#DF canonical-add(rx,ry)

     "{ ($rx$) is-canonical-float & ($ry$)
     is-canonical-float }"

     => ($construct-float (significand-part(rx) +
        significand-part(ry), exrad-part(rx))$)
        in-canonical-form #IF exrad-part(rx) =
```

```
   exrad-part(ry);

=> canonical-add (canonical-align(rx ,"to"
   exrad-part(ry)) ,"+" ry) #IF exrad-part(rx) >
   exrad-part(ry);

=> canonical-add(rx ,"+" canonical-align(ry ,"to"
   exrad-part(rx))) #OTHERWISE #.
```

"Floating point alignment is defined as multiplying the significand by 10 and decreasing the exrad by 1 until some desired exrad value is reached."

```
#DF canonical-align(r ,"to" e)

   "{r #IS <numeric-constant> & e #IS #INTEGER & e <
   exrad-part(r)}"

   => construct-float (significand-part(r) #CW
      ($exrad-part(r) - e$)zeros ,"and exrad" e) #.
```

```
#DF zeros(n)

   "{n #IS #INTEGER & n >= 0}"

   => #LEFT n #CHARACTERS-OF '0000000000' #IF n <= 10;

   => '0000000000' #CW ($n - 10$) zeros #OTHERWISE #.
```

"Floating point negation is defined in the obvious way."

```
#DF standard-negate(rx)

   "{rx #IS <numeric-constant> }"

   => construct-float (#NEG significand-part(($rx$)
      in-canonical-form), exrad-part(($rx$)
      in-canonical-form)) #.
```

"Floating point subtraction is defined in terms of negation and addition."

```
#DF standard-subtract(rx ,"-" ry)

     "{rx #IS <numeric-constant> & ry #IS <numeric-constant>
     }"

     => canonical-add( ($rx$) in-canonical-form,
        standard-negate(ry)) #.
```

"floating point multiply is defined as the canonical result of adding the two exrads to get the exrad of the result and multiplying the two significands to get the significand of the result."

```
#DF standard-multiply(rx,ry)

     "{rx #IS <numeric-constant> & ry #IS <numeric-constant>
     }"

     => canonical-multiply( ($rx$) in-canonical-form, ($ry$)
        in-canonical-form) #.


#DF canonical-multiply(rx,ry)

     "{($rx$)is-canonical-float & ($ry$)
     is-canonical-float}"

     =>  ($construct-float (significand-part(rx) *
        significand-part(ry), exrad-part(rx) +
        exrad-part(ry))$) in-canonical-form #.
```

"Floating point division is the most complex of the floating point operations. Leaving aside some detail, the division result is the canonical result of dividing the dividend significand by the divisor significand to get the result significand and subtracting the divisor exrad from the dividend exrad to get the result exrad. Division by zero will result in #UNDEFINED being returned. The extra complication with division is the desire to specify the precision of the result. As an example suppose 1 is divided by 3 to a precision of 2 significant digits. The result should be 33E-2. To do this, the significand is multiplied by 10^(p-1+dl) where p is the desired precision and dl is the length of the magnitude of the divisor. The exrad is

changed to e - p + 1 - dl. This shifting is to guarantee that the result of the division will have at least p digits. The division is performed and the result is then realigned to have exactly p digits of precision. Then that result is canonicalized."

```
#DF standard-divide(rdividend,rdivisor,precision)

     "{rdividend #IS <numeric-constant> & rdivisor #IS
     <numeric-constant> & precision #IS #INTEGER & precision
     > 0}"

     => canonical-divide( ($rdividend$) in-canonical-form,
        ($rdivisor$) in-canonical-form, precision) #.


#DF canonical-divide(rdividend,rdivisor,precision)

     "{($rdividend$) is-canonical-float & ($rdivisor$)
     is-canonical-float & precision #IS #INTEGER & precision
     > 0}"

     => #UNDEFINED #IF significand-part(rdivisor) = 0;

     => ($precision-limited ((significand-part(rdividend)
        #CW ($precision - 1 +
        divisor-length(rdivisor)$)zeros) /
        significand-part(rdivisor), exrad-part(rdividend) -
        precision + 1 - divisor-length(rdivisor) -
        exrad-part(rdivisor) ,"limited to" precision)$)
        in-canonical-form #OTHERWISE #.


#DF divisor-length(r)

     "{($r$) is-canonical-float}"

     => #LENGTH (#ABS (significand-part(r))) #.

     "precision-limited constructs a non-canonical floating
     point number whose significand has a specified
     precision from another number (represented by m and e)
     with a larger precision. This is done using truncation
     and not rounding."


#DF precision-limited(m,e ,"limited to" precision)
```

```
     "{m #IS #INTEGER & e #IS #INTEGER & p #IS #INTEGER & p
     > 0 & #LENGTH(#ABS(m)) >= p}"

     => construct-float (sign-string(m) #CW (#LEFT precision
        #CHARACTERS-OF #ABS(m)) ,"and exrad" e +
        #LENGTH(#ABS(m)) - precision) #.
```

"The sign string of a number is the string representing its sign. The sign string of a negative number is '-', zero or positive has a #NIL sign string"

```
#DF sign-string(r)

     "{r #IS <numeric-constant> }"

     => '-' #IF first-character-in (r) #EQW '-';

     => #NIL #OTHERWISE #.



     "COMMENT: A partial set of relational operators is
     defined on floating point numbers."


#DF is-standard-zero(n)

     "{ n #IS <numeric-constant> }"

     => #TRUE #IFF significand-part( ($n$)
        in-canonical-form) = 0 #.


#DF is-standard-negative(n)

     "{n #IS <numeric-constant> }"

     => #TRUE #IFF significand-part( ($n$)
        in-canonical-form) < 0 #.

     "The positive test does not include zero."


#DF is-standard-positive(n)
```


01/28/77

```
     "{ n #IS <numeric-constant>  }"

     => #TRUE #IFF significand-part( ($n$)
        in-canonical-form) > 0 #.


#DF are-standard-equal(a,b)

     "{ a #IS <numeric-constant>  & b #IS <numeric-constant>
     }"

     => #TRUE #IFF ($ standard-subtract(a,b) $)
        is-standard-zero #.


#DF is-standard-less-than(a ,"<" b)

     "{ a #IS <numeric-constant> & b #IS <numeric-constant>
     }"

     => #TRUE #IFF ($standard-subtract(a,b)$)
        is-standard-negative #.


#DF is-standard-greater-than(a ,">" b)

     "{ a #IS <numeric-constant> & b #IS <numeric-constant>
     }"

     => #TRUE #IFF ($standard-subtract(a,b)$)
        is-standard-positive #.
```

```
#DF is-string-expression(exp)

     "{ exp #IS <expression> }"

     => #TRUE #IFF exp #IS #CASE 1 #OF <expression> #.

#DF string-expression-of(exp)

     "{ ($exp$)is-string-expression }"

     => #SEG 1 #OF exp #.

#DF is-numeric-expression(exp)

     "{ exp #IS <expression> }"

     => #TRUE #IFF exp #IS  #CASE 2 #OF <expression> #.

#DF numeric-expression-of(exp)

     "{ ($exp$) is-numeric-expression #OR
        exp #IS <tab-call>}"

     => #SEG 1 #OF exp #IF ($exp$) is-numeric-expression ;

     => #SEG 5 #OF exp #IF exp #IS <tab-call> #.

#DF is-string-variable(exp)

     "{ exp #IS <string-expression> }"

     => #TRUE #IFF exp #IS #CASE 1 #OF <string-expression>
        #.

#DF string-variable-of(exp)

     "{ ($exp$)is-string-variable }"

     => #SEG 1 #OF exp #.
```

```
#DF is-string-constant(exp)

     "{ exp #IS <string-expression> }"

     => #TRUE #IFF exp #IS #CASE 2 #OF <string-expression>
        #.


#DF string-constant-of(exp)

     "{ ($exp$)is-string-constant }"

     => #SEG 1 #OF exp #.


#DF operand-1-of(x)

     "{ x #IS <expression> #U <numeric-expression> #U
     <positive-expression> #U <negation> #U <sum> #U
     <difference> #U <term> #U <product> #U <quotient> #U
     <factor> #U <involution> #U <relational-expression> #OR
     ($x$)is-parenthetical }"

     => #SEG 3 #OF x #IF x #IS <positive-expression> #U
        <negation> #OR ($x$)is-parenthetical;

     => #SEG 1 #OF x #OTHERWISE #.


#DF is-parenthetical(exp)

     "{ ($exp$)is-numeric-subnode }"

     => #TRUE #IFF exp #IS #CASE 4 #OF <primary> #.


#DF operand-2-of(x)

     "{ x #IS <sum> #U <difference> #U <product> #U
     <quotient> #U <involution> #U <relational-expression>
     }"

     => #SEG 5 #OF x #.


#DF is-numeric-defined-function-ref(ref)

     "{ref #IS <numeric-function-ref>}"
```

```
     => #TRUE #IFF ref #IS #CASE 1 #OF
        <numeric-function-ref> #.


#DF numeric-defined-function-ref-of(ref)

     "{($ref$) is-numeric-defined-function-ref}"

     => #SEG 1 #OF ref #.


#DF numeric-supplied-function-ref-of(ref)

     "{($ref$) is-numeric-supplied-function-ref}"

     => #SEG 1 #OF ref #.


#DF numeric-defined-function-name-of(dref)

     "{dref #IS <numeric-defined-function-ref>}"

     => #SEG 1 #OF dref #.


#DF has-an-argument(ref)

     "{ref #IS <numeric-defined-function-ref> #U
     <numeric-supplied-function-ref>}"

     => #TRUE #IFF ref #IS #CASE 2 #OF
        <numeric-defined-function-ref> #OR ref #IS
        <numeric-supplied-function-ref> #AND ref #IS-NOT
        #CASE 7 #OF <numeric-supplied-function-ref> #.


#DF argument-expression-of(ref)

     "{ref #IS <numeric-defined-function-ref> #U
     <numeric-supplied-function-ref>}"

     => #SEG 1 #OF (#SEG 3 #OF (#SEG 3 #OF ref)) #.


#DF def-statement-expression-of(def)

     "{def #IS def-statement}"
```

```
     => #SEG 7 #OF def #IF def #IS #CASE 1 #OF
        <def-statement> ;

     => #SEG 9 #OF def #OTHERWISE #.


#DF def-statement-parameter-of(def)

     "{ ($def$)is-def-statement-with-parameter}"

     => #SEG 3 #OF (#SEG 5 #OF def) #.


#DF is-def-statement-with-parameter(def)

     "{ def #IS #NODE}"

     => #TRUE #IFF  def #IS #CASE 2 #OF <def-statement> #.


#DF def-statement-name-of(def)

     "{ def #IS <def-statement> }"

     => #SEG 3 #OF def #.


#DF is-abs-function-ref(sref)

     "{sref #IS <numeric-supplied-function-ref>}"

     => #TRUE #IFF sref #IS #CASE 1 #OF
        <numeric-supplied-function-ref> #.


#DF is-atn-function-ref(sref)

     "{sref #IS <numeric-supplied-function-ref>}"

     => #TRUE #IFF sref #IS #CASE 2 #OF
        <numeric-supplied-function-ref> #.


#DF is-cos-function-ref(sref)

     "{sref #IS <numeric-supplied-function-ref>}"
```

```
     => #TRUE #IFF sref #IS #CASE 3 #OF
        <numeric-supplied-function-ref> #.


#DF is-exp-function-ref(sref)

     "{sref #IS <numeric-supplied-function-ref>}"

     => #TRUE #IFF sref #IS #CASE 4 #OF
        <numeric-supplied-function-ref> #.


#DF is-int-function-ref(sref)

     "{sref #IS <numeric-supplied-function-ref>}"

     => #TRUE #IFF sref #IS #CASE 5 #OF
        <numeric-supplied-function-ref> #.


#DF is-log-function-ref(sref)

     "{sref #IS <numeric-supplied-function-ref>}"

     => #TRUE #IFF sref #IS #CASE 6 #OF
        <numeric-supplied-function-ref> #.


#DF is-rnd-function-ref(sref)

     "{sref #IS <numeric-supplied-function-ref>}"

     => #TRUE #IFF sref #IS #CASE 7 #OF
        <numeric-supplied-function-ref> #.


#DF is-sgn-function-ref(sref)

     "{sref #IS <numeric-supplied-function-ref>}"

     => #TRUE #IFF sref #IS #CASE 8 #OF
        <numeric-supplied-function-ref> #.


#DF is-sin-function-ref(sref)

     "{sref #IS <numeric-supplied-function-ref>}"
```

```
     => #TRUE #IFF sref #IS #CASE 9 #OF
        <numeric-supplied-function-ref> #.


#DF is-sqr-function-ref(sref)

     "{sref #IS <numeric-supplied-function-ref>}"

     => #TRUE #IFF sref #IS #CASE 10 #OF
        <numeric-supplied-function-ref> #.


#DF is-tan-function-ref(sref)

     "{sref #IS <numeric-supplied-function-ref>}"

     => #TRUE #IFF sref #IS #CASE 11 #OF
        <numeric-supplied-function-ref> #.


#DF nameable-part-of(node)

     "{ node #IS <variable> #U <control-variable> #U
     <string-variable> #U <numeric-variable> #U
     <simple-numeric-variable> #U <numeric-array-element>}"

     => node #IF node #IS <string-variable> #U
        <simple-numeric-variable> #U <numeric-array-element>
        ;

     => #SEG 1 #OF node #IF node #IS <numeric-variable> ;

     => nameable-part-of(#SEG 1 #OF node) #IF node #IS
        <control-variable> #U <variable> #.


#DF numeric-array-name-of(node)

     "{ node #IS <numeric-array-element> #U
     <array-declaration> }"

     => #SEG 1 #OF node #.


#DF subscript-part-of(node)

     "{ node #IS <numeric-array-element> }"
```

```
     => #SEG 3 #OF node #.


#DF bounds-part-of(node)

     "{ node #IS &lt;array-declaration&gt; }"

     => #SEG 5 #OF node #.


#DF first-dimension-bound-of(b)

     "{ b #IS &lt;bounds&gt; }"

     => #SEG 1 #OF b #.


#DF has-one-dimension(b)

     "{ b #IS &lt;bounds&gt; #U &lt;subscript&gt; }"

     => #TRUE #IFF b #IS #CASE 1 #OF &lt;bounds&gt; #OR b #IS
        #CASE 1 #OF &lt;subscript&gt; #.


#DF second-dimension-bound-of(b)

     "{ ($b$) has-two-dimensions }"

     => #SEG 5 #OF b #.


#DF has-two-dimensions(b)

     "{ b #IS &lt;bounds&gt; #U &lt;subscript&gt; }"

     => #TRUE #IFF b #IS #CASE 2 #OF &lt;bounds&gt; #OR b #IS
        #CASE 2 #OF &lt;subscript&gt; #.


#DF first-dimension-of(s)

     "{ s #IS &lt;subscript&gt; }"

     => #SEG 3 #OF s #.


#DF second-dimension-of(s)
```

```
     "{ ($s$) has-two-dimensions }"

     => #SEG 7 #OF s #.


#DF option-base-of(opt)

     "{ opt #IS <option-statement> }"

     => #SEG 3 #OF opt #.


#DF relation-of (rel-exp)

     "{rel-exp #IS <relational-expression> }"

     => #SEG 3 #OF rel-exp #.


#DF is-numeric-relational-expression (rel-exp)

     "{rel-exp #IS <relational-expression> }"

     => #TRUE #IFF rel-exp #IS #CASE 1 #OF
        <relational-expression> #.


#DF is-string-relational-expression (rel-exp)

     "{rel-exp #IS <relational-expression> }"

     => #TRUE #IFF rel-exp #IS #CASE 2 #OF
        <relational-expression> #.


#DF relational-expression-of (stmt)

     "{stmt #IS <if-then-statement> }"

     => #SEG 3 #OF stmt #.


#DF line-number-part-of (ln )

     "{ln #IS <line> }"

     => #SEG 1 #OF ln #.
```

```
#DF destination-line-number-of (stmt)

     "{stmt #IS <goto-statement> #U <gosub-statement> #U
     <if-then-statement> }"

     => last-seg-of (stmt) #.


#DF index-expression-of (stmt)

     "{stmt #IS <on-goto-statement> }"

     => #SEG 3 #OF stmt #.


#DF last-seg-of (nx)

     "{#SEG-COUNT (nx) > 0 }"

     => #SEG (#SEG-COUNT (nx)) #OF nx #.


#DF is-non-executable(stmt)

     "{stmt #EQ current-statement}"

     => #TRUE #IFF stmt #IS <data-statement>
                        #U <def-statement>
                        #U <dimension-statement>
                        #U <option-statement>
                        #U <randomize-statement>
                        #U <remark-statement> #.


#DF is-simple-control-statement(stmt)

     "{stmt #EQ current-statement}"

     => #TRUE #IFF stmt #IS <goto-statement>
                        #U <if-then-statement>
                        #U <on-goto-statement>
                        #U <return-statement> #.


#DF control-variable-in(stmt)
```

```
     "{stmt #IS <for-statement> #U <next-statement> }"

     => #SEG 3 #OF stmt #.

#DF initial-value-part-of-for(stmt)

     "{stmt #IS <for-statement> }"

     => #SEG 1 #OF (#SEG 7 #OF stmt) #.

#DF limit-part-of-for(stmt)

     "{stmt #IS <for-statement> }"

     => #SEG 1 #OF (#SEG 11 #OF stmt) #.

#DF increment-part-of-for(stmt)

     "{stmt #IS <for-statement> }"

     => #SEG 1 #OF (#SEG 15 #OF stmt) #.

#DF is-quoted-string(d)

     "{d #IS <datum> }"

     => #TRUE #IFF d #IS #CASE 1 #OF <datum> #.

#DF is-numeric-variable(v)

     "{v #IS <variable> }"

     => #TRUE #IFF v #IS #CASE 1 #OF <variable> #.

#DF left-hand-side-of(stmt)

     "{stmt #IS <numeric-let-statement> #U
     <string-let-statement> }"

     => #SEG 3 #OF stmt #.
```

```
#DF right-hand-side-of(stmt)

     "{stmt #IS <numeric-let-statement> #U
     <string-let-statement> }"

     => #SEG 7 #OF stmt #.


#DF is-not-a-control-statement(stmt)

     "{stmt #EQ current-statement}"

     => #TRUE #IFF stmt #IS <input-statement>
                            #U <numeric-let-statement>
                            #U <string-let-statement>
                            #U <print-statement>
                            #U <read-statement>
                            #U <restore-statement>
                   #OR ($stmt$) is-non-executable #.


#DF statement-part-of(ln)

     "{ln #IS <line> }"

     => #SEG 1 #OF (#SEG 3 #OF ln) #.
```

"In order for a specification of an implementation of BASIC to be complete, The implementor must define his implementation dependent number representation and the implementation dependent arithmetic operations upon those numbers. The following definition is meant to serve as a guide for other implementors. This definition was chosen to use the minimal parameters specified by the BASIC standard. In particular the base machine is considered to use decimal floating point numbers with six decimal digits in the significand and two decimal digits in the exrad. The range of the magnitude of legal values is 1F38 down to 1E-38 (inclusive at both boundaries).

It should be noted that only one implementation-precision is defined. It is assumed that the actual precision of the 'machine' is the same as the precision of the results of arithmetic operators and the result of converting a BASIC constant to an implementation-number."

"Define the legal form of an implementation number. For purposes of this sample implementation, we choose to use the floating point definitions defined by the standard representation and the operations defined upon that representation. However, we will modify this representation to require that the numbers have significands of no more than six digits. It is strongly recommended that the reader be familiar with the operation of the standard floating point functions."

```
#DF is-implementation-number(n)

     => #TRUE #IFF ($n$)is-canonical-float &
        #LENGTH(#ABS(significand-part(n))) <=
        implementation-precision & #NOT ($standard-abs(n)$)
        is-an-overflow & #NOT ($standard-abs(n)$)
        is-an-underflow #.
```

"For many parts of the basic specification, it is important to know if an implementation number represents an integer. Because an implementation number is a canonical number, it cannot have any trailing zeros in its significand.

Therefore, it can only be an integer if its exrad is greater than or equal to zero."

```
#DF is-implementation-integer(n)

     "{ ($n$)is-implementation-number }"

     => #TRUE #IFF exrad-part(n) >= 0 #.

     "Certain auxiliary DFs must be defined to operate on a
     superset of the implementation-numbers.  In particular,
     those DFs which do range checking and limiting of
     precision."

     "Define a function to test for an implementation number
     greater than plus infinity or less than minus
     infinity."

#DF is-an-overflow(n)

     "{($n$) is-standard-float}"

     => #TRUE #IFF ($implementation-infinity ,
        standard-abs(n)$) is-standard-less-than #.

"Define an underflow test."

#DF is-an-underflow(n)

     "{($n$) is-standard-float}"

     => #TRUE #IFF  #NOT ($n$)is-standard-zero &
        ($standard-abs(n) , implementation-infinitesimal$)
        is-standard-less-than #.

"Define a function to truncate the significand of
 a number to the implementation-precision, with proper
exponent adjustment."

#DF limited-to-implementation-precision(n)
```

```
    "{($n$) is-standard-float}"

    => n #IF #LENGTH(#ABS(significand-part(n))) <=
       implementation-precision;

    => construct-float( sign-string(significand-part(n))
       #CW (#LEFT implementation-precision #CHARACTERS-OF
       #ABS(significand-part(n))), exrad-part(n) + #LENGTH
       (#ABS (significand-part(n))) -
       implementation-precision) #OTHERWISE #.


"Define whether the implementation detects underflow for the
purpose of reporting a non-fatal error."


#DF underflow-is-a-detected-non-fatal-error

    => #TRUE #.


    "Each implementation arithmetic operator requires a
    group of related implementation functions to test for
    overflow, underflow, etc, and one to perform the
    operation."


#DF results-in-negate-overflow(a)

    "{ ($a$)is-implementation-number }"

    => #FALSE #.


#DF non-fatal-negate-overflow-error-report

    => non-fatal-error('negate overflow') #.


#DF negate-overflow-result-sign(a)

    "{ ($a$) is-implementation-number & #NOT
    implementation-equals-test(a,implementation-zero) }"

    => '-' #IF
       implementation-greater-than-test(a,implementation-zero);
```

```
     => '+' #IF
        implementation-less-than-test(a,implementation-zero)
        #.


#DF results-in-negate-underflow(a)

     "{ ($a$)is-implementation-number }"

     => #FALSE #.


#DF non-fatal-negate-underflow-error-report

     => non-fatal-error('negate underflow') #.


#DF implementation-negate(a)

     "{ ($a$)is-implementation-number }"

     => ($standard-negate(a)$)
        limited-to-implementation-precision #.

     "The techniques used in implementation of addition and
     especially in underflow and overflow detection are
     quite general but are not strictly similar to the way
     in which a real machine might do things.  In
     particular, since large precision operations are
     available, the easiest way to test for overflow is to
     perform the exact addition and test the result for out
     of range."


#DF results-in-add-overflow(a,b)

     "{ ($a$)is-implementation-number &
     ($b$)is-implementation-number }"

     => #TRUE #IFF ($ standard-add(a,b) $)is-an-overflow #.


#DF non-fatal-add-overflow-error-report

     => non-fatal-error('add overflow') #.


#DF add-overflow-result-sign(a,b)
```

```
     "{ ($a$) is-implementation-number & #NOT
     implementation-equals-test(a,implementation-zero) &
     ($b$) is-implementation-number & #NOT
     implementation-equals-test(b,implementation-zero) }"

     => '+' #IF
        implementation-greater-than-test(a,implementation-zero);

     => '-' #IF
        implementation-less-than-test(a,implementation-zero)
        #.


#DF results-in-add-underflow(a,b)

     "{ ($a$)is-implementation-number &
     ($b$)is-implementation-number }"

     => #TRUE #IFF ($ standard-add(a,b) $)is-an-underflow #.


#DF non-fatal-add-underflow-error-report

     => non-fatal-error('add underflow') #.


#DF implementation-add(a,b)

     "{ ($a$)is-implementation-number &
     ($b$)is-implementation-number }"

     => ($standard-add(a,b)$)
        limited-to-implementation-precision #.


#DF results-in-subtract-overflow(a,b)

     "{ ($a$)is-implementation-number &
     ($b$)is-implementation-number }"

     => #TRUE #IFF
        ($standard-subtract(a,"-"b)$)is-an-overflow #.


#DF non-fatal-subtract-overflow-error-report

     => non-fatal-error('subtract-overflow') #.
```

```
#DF subtract-overflow-result-sign(a,b)

      "{ ($a$) is-implementation-number & #NOT
      implementation-equals-test(a,implementation-zero) &
      ($b$) is-implementation-number & #NOT
      implementation-equals-test(b,implementation-zero) }"

      => '+' #IF
         implementation-greater-than-test(a,implementation-zero);

      => '-' #IF
         implementation-less-than-test(a,implementation-zero)
         #.

#DF results-in-subtract-underflow(a,b)

      "{ ($a$)is-implementation-number &
      ($b$)is-implementation-number }"

      => #TRUE #IFF
         ($standard-subtract(a,"-"b)$)is-an-underflow #.

#DF non-fatal-subtract-underflow-error-report

      => non-fatal-error('subtract underflow') #.

#DF implementation-subtract(a,b)

      "{ ($a$)is-implementation-number &
      ($b$)is-implementation-number }"

      => ($standard-subtract(a,b)$)
         limited-to-implementation-precision #.

#DF results-in-multiply-overflow(a,b)

      "{ ($a$)is-implementation-number &
      ($b$)is-implementation-number }"

      => #TRUE #IFF ($standard-multiply(a,b)$)is-an-overflow
         #.
```

```
#DF non-fatal-multiply-overflow-error-report

     => non-fatal-error('multiply overflow') #.


#DF multiply-overflow-result-sign(a,b)

     "{ ($a$) is-implementation-number & #NOT
     implementation-equals-test(a,implementation-zero) &
     ($b$) is-implementation-number & #NOT
     implementation-equals-test(b,implementation-zero) }"

     => '+' #IF
        implementation-greater-than-test(a,implementation-zero)
        #IFF
        implementation-greater-than-test(b,implementation-zero);

     => '-' #IF
        implementation-less-than-test(a,implementation-zero)
        #IFF
        implementation-greater-than-test(b,implementation-zero)
        #.


#DF results-in-multiply-underflow(a,b)

     "{ ($a$)is-implementation-number &
     ($b$)is-implementation-number }"

     => #TRUE #IFF ($standard-multiply(a,b)$)
        is-an-underflow #.


#DF non-fatal-multiply-underflow-error-report

     => non-fatal-error('multiply underflow') #.


#DF implementation-multiply(a,b)

     "{ ($a$)is-implementation-number &
     ($b$)is-implementation-number }"

     => ($standard-multiply(a,b)$)
        limited-to-implementation-precision #.
```

```
#DF non-fatal-divide-by-zero-error-report

      => non-fatal-error('division by zero') #.


#DF divide-by-zero-result-sign(numerator)

      "{ ($numerator$) is-implementation-number }"

      => '+' #IF
         implementation-greater-than-test(numerator,implementation-zero);

      => '+' #IF
         implementation-equals-test(numerator,implementation-zero);

      => '-' #IF
         implementation-less-than-test(numerator,implementation-zero)
         #.


#DF results-in-divide-overflow(a,b)

      "{ ($a$)is-implementation-number &
      ($b$)is-implementation-number }"

      => #TRUE #IFF ($standard-divide(a,b ,"to"
         implementation-precision)$) is-an-overflow #.


#DF non-fatal-divide-overflow-error-report

      => non-fatal-error('divide overflow') #.


#DF divide-overflow-result-sign(a,b)

      "{ ($a$) is-implementation-number & #NOT
      implementation-equals-test(a,implementation-zero) &
      ($b$) is-implementation-number & #NOT
      implementation-equals-test(b,implementation-zero) }"

      => '+' #IF
         implementation-greater-than-test(a,implementation-zero)
         #IFF
         implementation-greater-than-test(b,implementation-zero);

      => '-' #IF
         implementation-less-than-test(a,implementation-zero)
```

```
     #IFF
     implementation-not-less-test(b,implementation-zero)
     #.


#DF results-in-divide-underflow(a,b)

    "{ ($a$)is-implementation-number &
    ($b$)is-implementation-number }"

    => #TRUE #IFF ($standard-divide(a,b ,"to"
       implementation-precision)$) is-an-underflow #.


#DF non-fatal-divide-underflow-error-report

    => non-fatal-error('divide underflow') #.


#DF implementation-divide(a,b)

    "{ ($a$)is-implementation-number &
    ($b$)is-implementation-number }"

    => standard-divide(a,b ,"to" implementation-precision)
       #.

    "The definition of involution is very difficult.  In
    order to handle the function, an external function is
    postulated which, for suitably restricted arguments,
    can return a result of more precision than
    implementation-precision.  The external function will
    never return zero.
         For the purposes of this example implementation,
    the domain of the  involution function is split into
    several parts.  This allows at least some test cases to
    involve the involution operator without involving the
    external function.  With that in mind, 0^x for any x is
    defined as 1.  x^0 for any x ia also 1.  Further, x^y
    for 0<y<=integer-exponentiation-limit and y an integer
    is performed using multiplication."


#DF special-involute(a,b)

    "{ ($a$)is-implementation-number &
    ($b$)is-implementation-number }"
```

```
     => implementation-one #IF standard-sign(a) = 0 #OR
        standard-sign(b) = 0;

     => standard-multiply(a, special-involute(a,
        standard-subtract(b, implementation-one))) #IF ($b$)
        is-implementation-integer &
        ($b$)is-in-integer-exponentiation-limit;

     => ($#EXTERNAL-CALL-OF 'involute' #WITH-ARGUMENT
        (\($a$) in-external-format, ($b$)
        in-external-format\)$)
        converted-from-external-format #OTHERWISE #.


#DF is-in-integer-exponentiation-limit(n)

     "{($n$)is-implementation-number}"

     => #TRUE #IFF #NOT ($n, integer-exponentiation-limit$)
        is-standard-greater-than #.


#DF integer-exponentiation-limit

     => '5E0' #.


#DF non-fatal-zero-involuted-to-negative-error-report

     => non-fatal-error('zero involuted to negative') #.


#DF results-in-involute-overflow(a,b)

     "{ ($a$)is-implementation-number &
     ($b$)is-implementation-number }"

     => #TRUE #IFF ($special-involute(a,b)$) is-an-overflow
        #.


#DF non-fatal-involute-overflow-error-report

     => non-fatal-error('involute overflow') #.


#DF involute-overflow-result-sign(a,b)
```

```
     "{ ($a$) is-implementation-number & #NOT
     implementation-equals-test(a,implementation-zero) &
     ($b$) is-implementation-number & #NOT
     implementation-equals-test(b,implementation-zero) }"

     => '+' #IF
        implementation-greater-than-test(a,implementation-zero)
        #OR ($b$) is-even-integer;

     => '-' #IF
        implementation-less-than-test(a,implementation-zero)
        & ($b$) is-odd-integer #.


#DF is-even-integer(n)

     "{ ($n$) is-implementation-number }"

     => residue( ($n$) converted-to-semanol-integer
        ,"modulo" 2) = 0 #IF ($n$) is-implementation-integer
        ;

     => #FALSE #OTHERWISE #.


#DF is-odd-integer(n)

     "{ ($n$) is-implementation-number }"

     => #NOT ($n$) is-even-integer #.


#DF results-in-involute-underflow(a,b)

     "{ ($a$)is-implementation-number &
     ($b$)is-implementation-number }"

     => #TRUE #IFF ($special-involute(a,b)$) is-an-underflow
        #.


#DF non-fatal-involute-underflow-error-report

     => non-fatal-error('involute underflow') #.


#DF implementation-involute(a,b)
```

```
     "{ ($a$)is-implementation-number &
     ($b$)is-implementation-number }"

     => ($special-involute(a,b)$)
        limited-to-implementation-precision #.


     "In general, the numeric-supplied functions will not be
     defined in the body of this SEMANOL program.  Instead,
     they utilize externally defined functions to calculate
     the results.  The external functions expect their
     arguments in some particular form and return a result
     in some particular form.  Conversion to and from this
     external form is accomplished by the functions
     in-external-format and converted-from-external-format."

#DF implementation-arctangent-function(n)

     "{($n$) is-implementation-number}"

     => ($#EXTERNAL-CALL-OF 'ATN' #WITH-ARGUMENT (\ ($n$)
        in-external-format \)$)
        converted-from-external-format #.

#DF implementation-cosine-function(n)

     "{($n$) is-implementation-number}"

     => ($#EXTERNAL-CALL-OF 'COS' #WITH-ARGUMENT (\ ($n$)
        in-external-format \)$)
        converted-from-external-format #.

#DF results-in-exponential-function-overflow(n)

     "{($n$) is-implementation-number}"

     => #TRUE #IFF
        ($implementation-exponential-function(n)$)
        is-an-overflow #.

#DF non-fatal-exponential-function-overflow-error-report
```

```
     => non-fatal-error('EXP overflow') #.


#DF exponential-function-result-sign(n)

     "{($n$) is-implementation-number}"

     => '+' #.


#DF results-in-exponential-function-underflow(n)

     "{($n$) is-implementation-number}"

     => #TRUE #IFF
        ($implementation-exponential-function(n)$)
        is-an-underflow #.


#DF non-fatal-exponential-function-underflow-error-report

     => non-fatal-error('EXP underflow') #.


#DF implementation-exponential-function(n)

     "{($n$) is-implementation-number}"

     => ($#EXTERNAL-CALL-OF 'EXP' #WITH-ARGUMENT (\ ($n$)
        in-external-format \)$)
        converted-from-external-format #.


#DF implementation-integer-function(n)

     "{($n$) is-implementation-number}"

     => ($#EXTERNAL-CALL-OF 'INT' #WITH-ARGUMENT (\ ($n$)
        in-external-format \)$)
        converted-from-external-format #.


#DF implementation-logarithm-function(n)

     "{($n$) is-implementation-number}"

     => ($#EXTERNAL-CALL-OF 'LOG' #WITH-ARGUMENT (\ ($n$)
        in-external-format \)$)
```

```
        converted-from-external-format #.


#DF implementation-random-function(r)

      "{r #IS #BOOLEAN}"

      => #EXTERNAL-CALL-OF 'RND' #WITH-ARGUMENT (\'TRUE'\)
         #IF r #EQ #TRUE;

      => #EXTERNAL-CALL-OF 'RND' #WITH-ARGUMENT (\'FALSE'\)
         #OTHERWISE #.


#DF implementation-sine-function(n)

      "{($n$) is-implementation-number}"

      => ($#EXTERNAL-CALL-OF 'SIN' #WITH-ARGUMENT (\ ($n$)
         in-external-format \)$)
         converted-from-external-format #.


#DF implementation-square-root-function(n)

      "{($n$) is-implementation-number}"

      => ($#EXTERNAL-CALL-OF 'SQR' #WITH-ARGUMENT (\ ($n$)
         in-external-format \)$)
         converted-from-external-format #.


#DF results-in-tangent-function-overflow(n)

      "{($n$) is-implementation-number}"

      => #TRUE #IFF ($implementation-tangent-function(n)$)
         is-an-overflow #.


#DF non-fatal-tangent-function-overflow-error-report

      => non-fatal-error('TAN overflow') #.


#DF tangent-function-result-sign(n)

      "{($n$) is-implementation-number}"
```

```
     => '-' #IF ($implementation-tangent-function(n)$)
        is-standard-negative;

     => '+' #OTHERWISE #.


#DF implementation-tangent-function(n)

     "{($n$) is-implementation-number}"

     => ($#EXTERNAL-CALL-OF 'EXP' #WITH-ARGUMENT (\ ($n$)
        in-external-format \)$)
        converted-from-external-format #.
```

"A set of implementation dependent relations is defined by BASIC. The implementation independent portion of the BASIC specification assumes that the implementation dependent relations will return a value in [#TRUE,#FALSE].

The BASIC standard is silent on the nature of the relations. In particular, It is assumed that no error conditions can occur in relational testing. If the testing is done using subtraction, this may not be the case since an overflow or underflow could occur. We choose to assume that this form of comparison is not legal and that no errors can occur."

```
#DF implementation-equals-test(a,b)

     "{ ($a$)is-implementation-number &
     ($b$)is-implementation-number }"

     => ($a,b$)are-standard-equal #.


#DF implementation-not-equals-test(a,b)

     "{ ($a$)is-implementation-number &
     ($b$)is-implementation-number }"

     => #NOT ($a,b$)are-standard-equal #.


#DF implementation-less-than-test(a,b)

     "{ ($a$)is-implementation-number &
     ($b$)is-implementation-number }"

     => ($a,b$)is-standard-less-than #.


#DF implementation-greater-than-test(a,b)

     "{ ($a$)is-implementation-number &
     ($b$)is-implementation-number }"

     => ($a,b$)is-standard-greater-than #.


#DF implementation-not-less-test(a,b)
```

```
    "{ ($a$)is-implementation-number &
    ($b$)is-implementation-number }"

    => #NOT ($a,b$)is-standard-less-than #.

#DF implementation-not-greater-test(a,b)

    "{ ($a$)is-implementation-number &
    ($b$)is-implementation-number }"

    => #NOT ($a,b$)is-standard-greater-than #.
```

"The implementor must define a function to convert from a BASIC constant into an implementation representation of that constant up to some precision, which in our case is the implementation-precision. A function must exist to convert from an implementation-representation to a BASIC constant."

"The numeric conversion routine has much in common with the arithmetic operators. It must have functions to check for conversion underflow and overflow. The method used in this implementation will only work correctly if rounding is performed before underflow or overflow tests."

```
#DF results-in-numeric-conversion-overflow(n)

     "{n #IS #STRING & n #IS <numeric-constant>}"

     => #TRUE #IFF ($conversion-rounded (($n$)
        in-canonical-form)$) is-an-overflow #.


#DF results-in-numeric-conversion-underflow(n)

     "{n #IS #STRING & n #IS <numeric-constant>}"

     => #TRUE #IFF ($conversion-rounded (($n$)
        in-canonical-form)$) is-an-underflow #.


#DF implementation-numeric-representation(n)

     "{n #IS <numeric-constant>
      & #NOT ($n$) results-in-numeric-conversion-overflow &
     #NOT ($n$) results-in-numeric-conversion-underflow}"

     => ($conversion-rounded (($n$) in-canonical-form)$)
        limited-to-implementation-precision #.


#DF conversion-rounded(n)

     "{ ($n$) is-canonical-float }"

     => n #IF #LENGTH(#ABS(significand-part(n))) <=
```

```
        implementation-precision;

     => ($n ,"and-p-of" implementation-precision$)
        rounded-to-p-digits #OTHERWISE #.


"Define the error report for numeric constant conversions
and the function to determine the result sign in a numeric
constant conversion overflow."

#DF non-fatal-numeric-constant-overflow-error-report

     => non-fatal-error('numeric constant conversion
        overflow') #.

#DF non-fatal-numeric-constant-underflow-error-report

     => non-fatal-error('numeric-constant conversion
        underflow') #.

#DF numeric-constant-overflow-result-sign(s)

     "{s #IS <numeric-constant> }"

     => '-' #IF sign-string(s) #EQW '-';

     => '+' #OTHERWISE #.


     "Rounding, as defined in the glossary of the BASIC
     standard, applies to machines of any radix.  For the
     purposes of this implementation we will perform decimal
     rounding by adding 0.5 and truncating the result."

#DF rounded-to-an-integer(r)

     "{($r$) is-implementation-number}"

     => ($standard-add(r, implementation-one-half)$)
        truncated-to-an-integer #.
```

```
#DF implementation-one-half

      => '5E-1' #.

      "Truncation to an integer has three cases.  The first
      case is that the number,n, is already an integer.  The
      second case is that n has a magnitude less than one.
      The third case is a number with at least one fractional
      digit and at least one integer digit."


#DF truncated-to-an-integer(r)

      "{($r$) is-implementation-number}"

      => r #IF ($r$) is-implementation-integer;

      => implementation-zero #IF
         #LENGTH(#ABS(significand-part(r))) <= #NEG
         exrad-part(r);

      => ($construct-float (#LEFT
         #LENGTH(significand-part(r)) - (#NEG exrad-part(r))
         #CHARACTERS-OF significand-part(r) ,"with integer
         exrad" 0)$) in-canonical-form #OTHERWISE #.



"The implementor is required to define a conversion function
from implementation numbers into standard form numbers (i.e.
BASIC numeric-constants).  This conversion must be exact,
which means that the following expression must be true for
all x in the set of implementation numbers.
      #DF conversion-to-standard-float-is-exact(x)
         ''{($x$) is-implementation-number}''
      => implementation-equals-test(
         implementation-numeric-representation(
         ($x$) converted-to-standard-float)
         ,''is-equal-to'' x) #.
      Since this example implementation already uses the
standard form for its representation,  the conversion to the
standard form is the trivial one."


#DF converted-to-standard-float(x)
```

```
"{($x$) is-implementation-number}"

=> x #.
```

```
#DF max-number-of-unreturned-gosubs

     => 10 #.

#DF max-number-of-for-blocks

     => 10 #.

#DF implementation-margin

     => 65 #.

#DF implementation-print-zone-width

     => 15 #.

#DF max-assignable-string-length

     => 18 #.

#DF implementation-significance-width

     => 3 #.

#PROC-DF fatal-error(msg)

     "{msg #IS #STRING}"

     #BEGIN

       #COMPUTE! #OUTPUT(msg #CW end-of-print-line-char)

       #COMPUTE! #OUTPUT('execution-terminated' #CW
       end-of-print-line-char)

       #COMPUTE! #STOP

     #END #.
```

```
#DF non-fatal-error(msg)

     "{msg #IS #STRING}"

     => #OUTPUT(msg #CW end-of-print-line-char) #.


#DF implementation-string-output-representation (str-rep)

     "{($str-rep$) is-implementation-string}"

     => str-rep #.

     "Define various number representation parameters and
     functions."


#DF implementation-precision

     => 3 #.


#DF implementation-zero

     => '0E0' #.


#DF implementation-one

     => '1E0' #.


#DF implementation-negative-one

     => '-1E0' #.


#DF implementation-infinity

     => '1E4' #.

     "Note that the standard seems to require that the
     magnitudes of positive and negative infinity be the
     same."


#DF implementation-negative-infinity
```

```
     => '-1E4' #.

     "Similarly define the infinitesimals."

#DF implementation-infinitesimal

     => '1E-4' #.

#DF implementation-negative-infinitesimal

     => '-1E-4' #.
```

```
#DF root-node(n)

      "{ n #IS #NODE }"

      => basic-program #.


#DF parent-node(n)

      "{ n #IS #NODE }"

      => #FIRST x #IN (#SEQUENCE-OF-NODES-IN basic-program)
         #SUCH-THAT( #THERE-EXISTS i:1<=i<=#SEG-COUNT(x)
         #SUCH-THAT( #SEG i #OF x #EQ n)) #.


#DF sequence-of-ancestors-of(n)

      "{ n #IS #NODE }"

      => #NILSEQ #IF n #EQ basic-program ;

      => #SUBSEQUENCE-OF-ELEMENTS x #IN
         (#SEQUENCE-OF-NODES-IN basic-program) #SUCH-THAT( n
         #IS-IN #SEQUENCE-OF-NODES-IN x) #OTHERWISE #.


#DF first-character-in(s)

      "{ s #IS #STRING }"

      => #LEFT 1 #CHARACTERS-OF s #.


#DF last-character-in(s)

      "{ s #IS #STRING }"

      => #RIGHT 1 #CHARACTERS-OF s #.


#DF reverse-sequence(seq)

      "{ seq #IS #SEQUENCE }"

      => seq #IF #LENGTH(seq) <= 1 ;
```

```
=> #LAST-ELEMENT-IN seq #CS reverse-sequence(
   #TERMINAL-SUBSEQ-OF-LENGTH #LENGTH(seq) - 1 #OF seq)
   #OTHERWISE #.
```

METRIC SYSTEM

BASE UNITS:

Quantity	Unit	SI Symbol	Formula
length	metre	m	...
mass	kilogram	kg	...
time	second	s	...
electric current	ampere	A	...
thermodynamic temperature	kelvin	K	...
amount of substance	mole	mol	...
luminous intensity	candela	cd	...

SUPPLEMENTARY UNITS:

plane angle	radian	rad	...
solid angle	steradian	sr	...

DERIVED UNITS:

Acceleration	metre per second squared	...	m/s
activity (of a radioactive source)	disintegration per second	...	(disintegration)/s
angular acceleration	radian per second squared	...	rad/s
angular velocity	radian per second	...	rad/s
area	square metre	...	m
density	kilogram per cubic metre	...	kg/m
electric capacitance	farad	F	A·s/V
electrical conductance	siemens	S	A/V
electric field strength	volt per metre	...	V/m
electric inductance	henry	H	V·s/A
electric potential difference	volt	V	W/A
electric resistance	ohm		V/A
electromotive force	volt	V	W/A
energy	joule	J	N·m
entropy	joule per kelvin	...	J/K
force	newton	N	kg·m/s
frequency	hertz	Hz	(cycle)/s
illuminance	lux	lx	lm/m
luminance	candela per square metre	...	cd/m
luminous flux	lumen	lm	cd·sr
magnetic field strength	ampere per metre	...	A/m
magnetic flux	weber	Wb	V·s
magnetic flux density	tesla	T	Wb/m
magnetomotive force	ampere	A	...
power	watt	W	J/s
pressure	pascal	Pa	N/m
quantity of electricity	coulomb	C	A·s
quantity of heat	joule	J	N·m
radiant intensity	watt per steradian	...	W/sr
specific heat	joule per kilogram-kelvin	...	J/kg·K
stress	pascal	Pa	N/m
thermal conductivity	watt per metre-kelvin	...	W/m·K
velocity	metre per second	...	m/s
viscosity, dynamic	pascal-second	...	Pa·s
viscosity, kinematic	square metre per second	...	m/s
voltage	volt	V	W/A
volume	cubic metre	...	m
wavenumber	reciprocal metre	...	(wave)/m
work	joule	J	N·m

SI PREFIXES:

Multiplication Factors	Prefix	SI Symbol
1 000 000 000 000 = 10^{12}	tera	T
1 000 000 000 = 10^{9}	giga	G
1 000 000 = 10^{6}	mega	M
1 000 = 10^{3}	kilo	k
100 = 10^{2}	hecto*	h
10 = 10^{1}	deka*	da
0.1 = 10^{-1}	deci*	d
0.01 = 10^{-2}	centi*	c
0.001 = 10^{-3}	milli	m
0.000 001 = 10^{-6}	micro	μ
0.000 000 001 = 10^{-9}	nano	n
0.000 000 000 001 = 10^{-12}	pico	p
0.000 000 000 000 001 = 10^{-15}	femto	f
0.000 000 000 000 000 001 = 10^{-18}	atto	a

* To be avoided where possible.

MISSION
of
Rome Air Development Center

RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C^3) activities, and in the C^3 areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

AMERICAN REVOLUTION BICENTENNIAL 1776-1976